Node.js

Aplicações web real-time com Node.js

CAIO RIBEIRO PEREIRA

Agradecimentos

Primeiramente, quero agradecer a Deus por tudo que fizeste em minha vida! Agradeço também ao meu pai e minha mãe pelo amor, força, incentivo e por todo apoio desde o meu início de vida. Obrigado por tudo e principalmente por estar ao meu lado em todos os momentos.

Um agradecimento especial a minha namorada Natália Santos, afinal começamos a namorar na mesma época que comecei este livro, e sua companhia, compreensão e incentivo foram essenciais para persistir neste projeto.

Agradeço a sra. Charlotte Bento de Carvalho, pelo apoio e incentivo nos meus estudos desde a escola até a minha formatura na faculdade.

Um agradecimento ao meu primo Cláudio Souza. Foi graças a ele que entrei nesse mundo da tecnologia. Ele foi a primeira pessoa a me apresentar o computador e me aconselhou anos depois a entrar em uma faculdade de TI.

Um agradecimento ao Bruno Alvares da Costa, Leandro Alvares da Costa e Leonardo Pinto, esses caras me apresentaram um mundo novo da área de desenvolvimento de *software*. Foram eles que me influenciaram a escrever um blog, a palestrar em eventos, a participar de comunidades e fóruns, e principalmente a nunca cair na zona de conforto, a aprender sempre. Foi uma honra trabalhar junto com eles em 2011. E hoje, mesmo muita coisa tendo mudado, ainda tenho a honra de trabalhar com o Leandro numa nova startup que já está virando uma empresa, que é a BankFacil.

Obrigado pessoal da editora Casa do Código, em especial ao Paulo Silveira e Adriano Almeida. Muito obrigado pelo suporte e pela oportunidade!

Obrigado galera da comunidade NodeBR. Seus *feedbacks* ajudaram a melhorar este livro e também agradeço a todos os leitores do blog Underground WebDev. Afinal a essência deste livro foi baseado nos posts sobre Node.js publicados lá.

Por último, obrigado você, prezado leitor, por adquirir este livro. Espero que este livro seja uma ótima referência para ti.

Comentários

Veja abaixo alguns comentários no blog Underground WebDev a respeito do conteúdo que você esta prestes a ler.

Parabéns pelo Post! Adorei, muito explicativo. A comunidade brasileira agradece.
– Rafael Henrique Moreira - virtualjoker@gmail.com - "http://nodebr.com"

Tive o prazer de trocar experiências e aprender muito com o Caio. Um cara singular à "instância", do típico nerd que abraça um problema e não desgruda até resolvê-lo. Obrigado pela ajuda durante nosso tempo trabalho e não vou deixar de acompanhar essas aulas. Parabéns!
– Magno Ozzyr - magno_ozzyr@hotmail.com

Digno de reconhecimento o empenho do Caio no projeto de contribuir com o desenvolvimento e propagação dessa tecnologia. Isso combina com o estilo ambicioso e persistente que sempre demonstrou no processo de formação. Sucesso! Continue compartilhando os frutos do seu trabalho para assim deixar sua marca na história da computação.
– Fernando Macedo - fernando@fmacedo.com.br - "http://fmacedo.com.br"

Ótimo conteúdo, fruto de muito trabalho e dedicação. Conheci o Caio ainda na faculdade, sempre enérgico, às vezes impaciente por causa de sua ânsia pelo novo. Continue assim buscando aprender mais e compartilhando o que você conhece com os outros. Parabéns pelo trabalho!
– Thiago Ferauche - thiago.ferauche@gmail.com

Wow, muito bacana Caio! Eu mesmo estou ensaiando para aprender Javascript e cia. Hoje trabalho mais com HTML/CSS, e essa ideia de "para Leigos" me interessa muito! Fico no aguardo dos próximos posts!! =)
– Marcio Toledo - mntoledo@gmail.com - "http://marciotoledo.com"

Caião, parabéns pela iniciativa, pelo trabalho e pela contribuição para a comunidade. Trabalhamos juntos e sei que você é uma pessoa extremamente dedicada e ansioso por novos conhecimentos. Continue assim e sucesso!
– Leonardo Pinto - leonardo.pinto@gmail.com

Caio, parabéns pelo curso e pelo conteúdo. É sempre bom contar com material de qualidade produzido no Brasil, pois precisamos difundir o uso de novas tecnologias e encorajar seu uso.
– Evaldo Junior - evaldojuniorbento@gmail.com - "http://evaldojunior.com.br"

Parabéns pela iniciativa! Acredito que no futuro você e outros façam mais cursos do mesmo, sempre buscando compartilhar o conhecimento pra quem quer aprender.
– Jadson Lourenço - "http://twitter.com/jadsonlourenco"

Sobre o autor

Figura 1: Caio Ribeiro Pereira

Sou *Web Developer* na *startup* BankFacil, minha experiência baseia-se no domínio dessa sopa de letrinhas: Node.js, Modular Javascript, Modular CSS, Ruby, Java, MongoDB, Redis, Agile, Filosofia Lean, Scrum, XP, Kanban e TDD.

Bacharel em Sistemas de Informação pela Universidade Católica de Santos, blogueiro nos tempos livres, apaixonado por programação, web, tecnologias, filmes e seriados.

Participante das comunidades:

- NodeBR: Comunidade Brasileira de Node.js
- MeteorBrasil: Comunidade Brasileira de Meteor
- DevInSantos: Grupo de Desenvolvedores de Software em Santos

Iniciei em 2011 como palestrante nos eventos DevInSantos e Exatec, abordando temas atuais sobre Node.js e Javascript.

Autor dos Blogs: Underground WebDev e Underground Linux.

Prefácio

As mudanças do mundo web

Tudo na web se trata de consumismo e produção de conteúdo. Ler ou escrever blogs, assistir ou enviar vídeos, ver ou publicar fotos, ouvir músicas e assim por diante. Isso fazemos naturalmente todos os dias na internet. E cada vez mais aumenta a necessidade dessa interação entre os usuários com os diversos serviços da web. De fato, o mundo inteiro quer interagir mais e mais na internet, seja através de conversas com amigos em chats, jogando games online, atualizando constantemente suas redes sociais ou participando de sistemas colaborativos. Esses tipos de aplicações requerem um poder de processamento extremamente veloz, para que seja eficaz a interação em tempo real entre cliente e servidor. E mais, isto precisa acontecer em uma escala massiva, suportando de centenas a milhões de usuários.

Então o que nós desenvolvedores precisamos fazer? Nós precisamos criar uma comunicação em tempo real entre cliente e servidor — que seja rápido, atenda muitos usuários ao mesmo tempo e utilize recursos de I/O (dispositivos de entrada ou saída) de forma eficiente. Qualquer pessoa com experiência desenvolvimento web sabe que o HTTP não foi projetado para suportar estes requisitos. E pior, infelizmente existem sistemas que os adotam de forma ineficiente e incorreta, implementando soluções *workaround* ("Gambiarras") que executam constantemente requisições assíncronas no servidor, mais conhecidas como *long-polling*. Para sistemas trabalharem em tempo real, servidores precisam enviar e receber dados utilizando comunicação bidirecional, ao invés de utilizar intensamente requisição e resposta do modelo HTTP através do *Ajax*. E também temos que manter esse tipo comunicação de forma leve e rápida para manter escalável, reutilizável e fácil de manter o desenvolvimento a longo prazo.

A quem se destina esse livro?

Esse livro é destinado aos desenvolvedores web, que tenham pelo menos conhecimentos básicos de Javascript e arquitetura cliente-servidor. Ter domínio desses

conceitos, mesmo que seja um conhecimento básico deles, será essencial para que a leitura desse livro seja de fácil entendimento.

Como devo estudar?

Ao decorrer da leitura serão apresentados diversos conceitos e códigos, para que você aprenda na prática toda a parte teórica do livro. A partir do capítulo 4 até o capítulo final, iremos desenvolver na prática um projeto web, utilizando os principais frameworks e aplicando as boas práticas de desenvolvimento Javascript para Node.js.

Sumário

Capítulo 1

Bem-vindo ao mundo Node.js

1.1 O problema das arquiteturas bloqueantes

Os sistemas para web desenvolvidos sobre plataforma **.NET, Java, PHP, Ruby ou Python** possuem uma característica em comum: eles paralisam um processamento enquanto utilizam um I/O no servidor. Essa paralisação é conhecida como modelo bloqueante *(Blocking-Thread)*. Em um servidor web podemos visualizá-lo de forma ampla e funcional. Vamos considerar que cada processo é requisição feita pelo usuário. Com o decorrer da aplicação, novos usuários vão acessando-a, gerando uma requisição no servidor. Um sistema bloqueante enfileira cada requisição e depois as processa, uma a uma, não permitindo múltiplos processamentos delas. Enquanto uma requisição é processada as demais ficam em espera, mantendo por um período de tempo uma fila de requisições ociosas.

Esta é uma arquitetura clássica, existente em diversos sistemas pelo qual possui um design ineficiente. É gasto grande parte do tempo mantendo uma fila ociosa enquanto é executado um I/O. Tarefas como enviar e-mail, consultar o banco de dados, leitura em disco, são exemplos de tarefas que gastam uma grande fatia desse

tempo, bloqueando o sistema inteiro enquanto não são finalizadas. Com o aumento de acessos no sistema, a frequência de gargalos serão mais frequentes, aumentando a necessidade de fazer um *upgrade* nos *hardwares* dos servidores. Mas *upgrade* das máquinas é algo muito custoso, o ideal seria buscar novas tecnologias que façam bom uso do *hardware* existente, que utilizem ao máximo o poder do processador atual, não o mantendo ocioso quando o mesmo realizar tarefas do tipo bloqueante.

1.2 E assim nasceu o Node.js

Figura 1.1: Logotipo do Node.js.

Foi baseado neste problema que, no final de 2009, Ryan Dahl com a ajuda inicial de 14 colaboradores criou o Node.js. Esta tecnologia possui um modelo inovador, sua arquitetura é totalmente *non-blocking thread* (não-bloqueante), apresentando uma boa performance com consumo de memória e utilizando ao máximo e de forma eficiente o poder de processamento dos servidores, principalmente em sistemas que produzem uma alta carga de processamento. Usuários de sistemas Node estão livres de aguardarem por muito tempo o resultado de seus processos, e principalmente não sofrerão de *dead-locks* no sistema, porque nada bloqueia em sua plataforma e desenvolver sistemas nesse paradigma é simples e prático.

Esta é uma plataforma altamente escalável e de baixo nível, pois você vai programar diretamente com diversos protocolos de rede e internet ou utilizar bibliotecas que acessam recursos do sistema operacional, principalmente recursos de sistemas baseado em Unix. O Javascript é a sua linguagem de programação, e isso foi possível graças à *engine* **Javascript V8**, a mesma utilizada no navegador *Google Chrome*.

1.3 Single-thread

Suas aplicações serão *single-thread*, ou seja, cada aplicação terá instância de um único processo. Se você esta acostumado a trabalhar com programação concorrente em plataforma *multi-thread*, infelizmente não será possível com Node, mas saiba que existem outras maneiras de se criar um sistema concorrente, como por exemplo, utilizando *clusters* (assunto a ser explicado no capítulo 9.2), que é um módulo nativo do Node.js e é super fácil de implementá-lo. Outra maneira é utilizar ao máximo a programação assíncrona. Esse será o assunto mais abordado durante o decorrer deste livro, pelo qual explicarei diversos cenários e exemplos práticos em que são executados em paralelo funções em *background* que aguardam o seu retorno através de funções de *callback* e tudo isso é trabalhado forma não-bloqueante.

1.4 Event-Loop

Node.js é orientado a eventos, ele segue a mesma filosofia de orientação de eventos do Javascript client-side; a única diferença é que não existem eventos de `click` do mouse, `keyup` do teclado ou qualquer evento de componentes HTML. Na verdade trabalhamos com eventos de I/O do servidor, como por exemplo: o evento `connect` de um banco de dados, um `open` de um arquivo, um `data` de um *streaming* de dados e muitos outros.

O *Event-Loop* é o agente responsável por escutar e emitir eventos no sistema. Na prática ele é um loop infinito que a cada iteração verifica em sua fila de eventos se um determinado evento foi emitido. Quando ocorre, é emitido um evento. Ele o executa e envia para fila de executados. Quando um evento está em execução, nós podemos programar qualquer lógica dentro dele e isso tudo acontece graças ao mecanismo de função *callback* do Javascript.

O design *event-driven* do Node.js foi inspirado pelos frameworks Event Machine do Ruby (http://rubyeventmachine.com) e Twisted do Python (http://twistedmatrix.com) . Porém, o *Event-loop* do Node é mais performático por que seu mecanismo é nativamente executado de forma não-bloqueante. Isso faz dele um grande diferencial em relação aos seus concorrentes que realizam chamadas bloqueantes para iniciar os seus respectivos *Event-loops*.

1.5 Instalação e configuração

Para configurar o ambiente Node.js, independente de qual sistema operacional você utilizar, as dicas serão as mesmas. É claro que os procedimentos serão diferentes para cada sistema (principalmente para o Windows, mas não será nada grave).

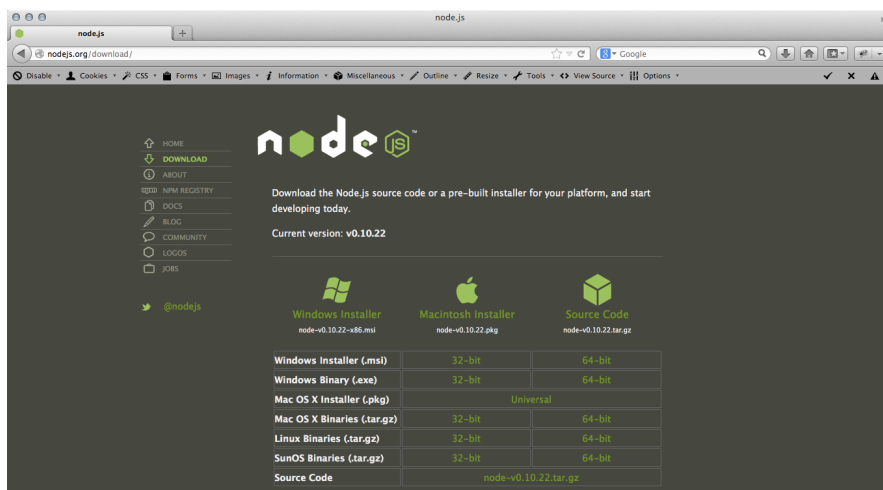
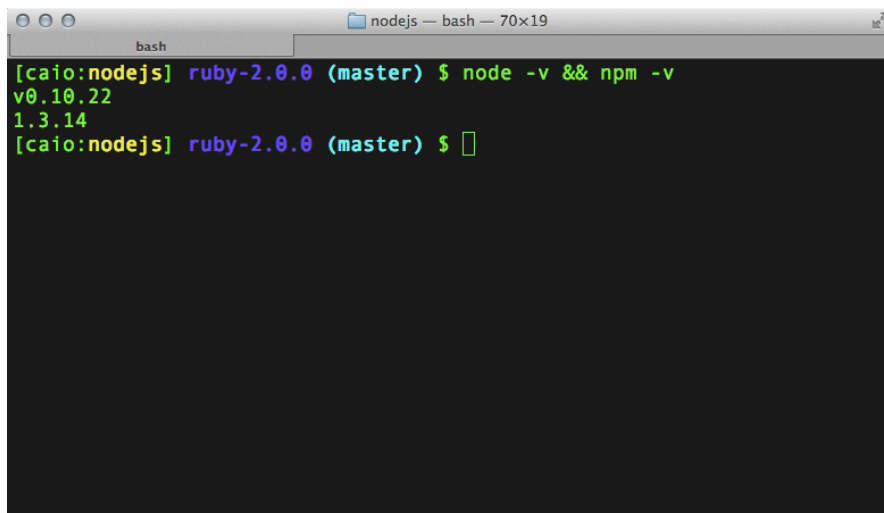



Figura 1.2: Página de Download do Node.js.

Instalando Node.js: Primeiro passo, acesse o site oficial: (http://nodejs.org) e clique em **Download**, para usuários do *Windows e MacOSX*, basta baixar os seus instaladores e executá-los normalmente. Para quem já utiliza *Linux* com *Package Manager* instalado, acesse esse link (https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager) que é referente as instruções sobre como instalá-lo em diferentes sistemas. Instale o Node.js de acordo com seu sistema, caso não ocorra problemas, basta abrir o seu terminal console ou prompt de comando e digitar o comando: `node -v && npm -v` para ver as respectivas versões do Node.js e NPM *(Node Package Manager)* que foram instaladas.

```
nodejs — bash — 70×19
bash
[caio:nodejs] ruby-2.0.0 (master) $ node -v && npm -v
v0.10.22
1.3.14
[caio:nodejs] ruby-2.0.0 (master) $
```

Figura 1.3: Versão do Node.js e NPM utilizada neste livro.

A última versão estável utilizada neste livro é **Node 0.10.22** e **NPM 1.3.14**. Alias é altamente recomendável utilizar esta versão ou superior, pois recentemente foi identificado uma vulnerabilidade de ataque *DoS* em versões anteriores, veja mais detalhes no blog oficial: (http://blog.nodejs.org/2013/10/22/cve-2013-4450-http-server-pipeline-flood-dos) Dica: Todo conteúdo deste livro será compatível com versões do Node.js **0.8.0** ou superiores. **Configurando ambiente de desenvolvimento:** Para configurá-lo basta adicionar uma variável de ambiente **NODE_ENV** no sistema operacional. Em sistemas *Linux ou OSX*, basta acessar com um editor de texto qualquer e em modo super user **(sudo)** o arquivo `.bash_profile` ou `.bashrc` e adicionar o seguinte comando: `export NODE_ENV='development'`. No Windows 7, o processo é um pouco diferente.

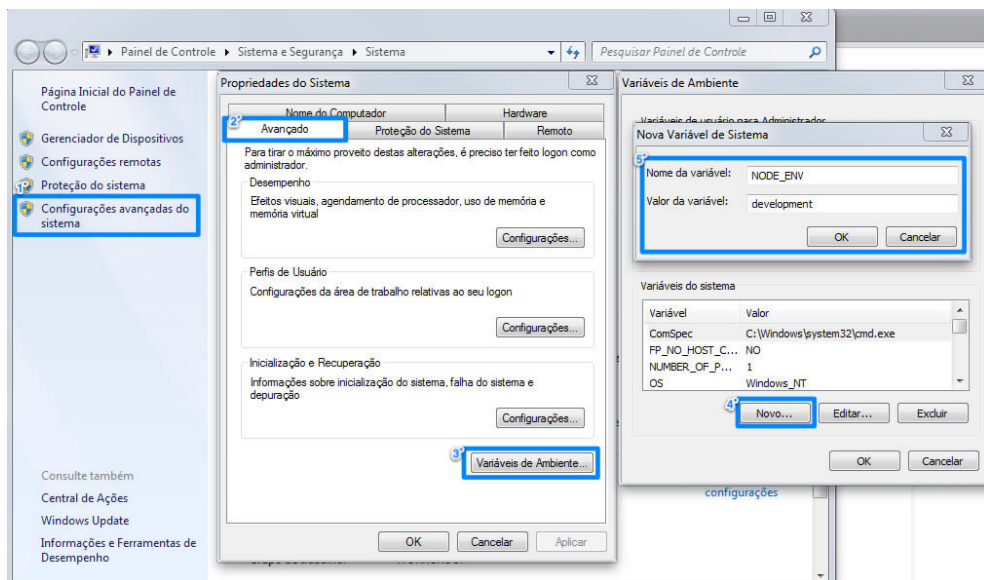
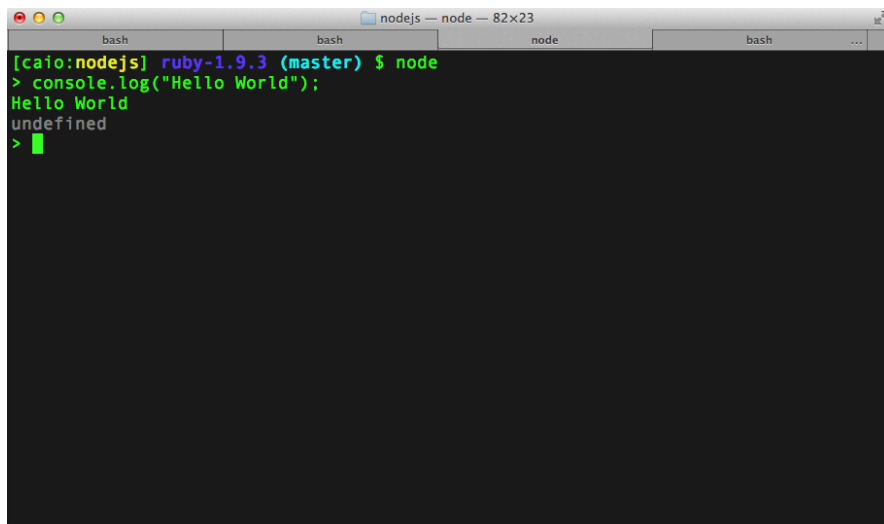



Figura 1.4: Configurando a variável NODE_ENV no Windows 7.

Clique com botão direito no ícone **Meu Computador** e selecione a opção **Propriedades**, no lado esquerdo da janela, clique no link **Configurações avançadas do sistema**. Na janela seguinte, acesse a aba **Avançado** e clique no botão **Variáveis de Ambiente...**, agora no campo **Variáveis do sistema** clique no botão **Novo...**, em nome da variável digite `NODE_ENV` e em valor da variável digite: `development`. Após finalizar essa tarefa reinicie seu computador para carregar essa variável no sistema operacional.

Rodando o Node: Para testarmos o ambiente, executaremos o nosso primeiro programa *Hello World*. Execute o comando: `node` para acessarmos o REPL *(Read-Eval-Print-Loop)* que permite executar código Javascript diretamente no terminal , digite `console.log("Hello World");` e tecle ENTER para executá-lo na hora.

Figura 1.5: Hello World via REPL do Node.js

1.6 Gerenciando módulos com NPM

Assim como o *Gems* do *Ruby* ou o *Maven* do *Java*, o Node.js também possui o seu próprio gerenciador de pacotes, ele se chama NPM *(Node Package Manager)*. Ele se tornou tão popular pela comunidade, que foi a partir da versão **0.6.0** do Node.js que ele se integrou no instalador do Node.js, tornando-se o gerenciador *default*. Isto simplificou a vida dos desenvolvedores na época, pois fez com que diversos projetos se convergissem para esta plataforma. Não listarei todos, mas apenas os comandos principais para que você tenha noções de como gerenciar módulos nele:

- `npm install nome_do_módulo`: instala um módulo no projeto.
- `npm install -g nome_do_módulo`: instala um módulo global.
- `npm install nome_do_módulo --save`: instala o módulo no projeto, atualizando o `package.json` na lista de dependências.
- `npm list`: lista todos os módulos do projeto.
- `npm list -g`: lista todos os módulos globais.
- `npm remove nome_do_módulo`: desinstala um módulo do projeto.

- `npm remove -g nome_do_módulo`: desinstala um módulo global.
- `npm update nome_do_módulo`: atualiza a versão do módulo.
- `npm update -g nome_do_módulo`: atualiza a versão do módulo global.
- `npm -v`: exibe a versão atual do npm.
- `npm adduser nome_do_usuário`: cria uma conta no npm, através do site (https://npmjs.org) .
- `npm whoami`: exibe detalhes do seu perfil público npm (é necessário criar uma conta antes).
- `npm publish`: publica um módulo no site do npm, é necessário ter uma conta antes.

1.7 Entendendo o package.json

Todo projeto Node.js é chamado de módulo, mas o que é um módulo? No decorrer da leitura, perceba que falarei muito sobre o termo módulo, biblioteca e framework, e na prática eles possuem o mesmo significado. O termo módulo surgiu do conceito de que a arquitetura do Node.js é modular. E todo módulo é acompanhado de um arquivo descritor, conhecido pelo nome de `package.json`.

Este arquivo é essencial para um projeto Node.js. Um `package.json` mal escrito pode causar bugs ou impedir o funcionamento correto do seu módulo, pois ele possui alguns atributos chaves que são compreendidos pelo Node.js e NPM.

No código abaixo apresentarei um `package.json` que contém os principais atributos para descrever um módulo:

```
{
    "name": "meu-primero-node-app",
    "description": "Meu primeiro app em Node.js",
    "author": "Caio R. Pereira <caio@email.com>",
    "version": "1.2.3",
    "private": true,
    "dependencies": {
        "modulo-1": "1.0.0",
        "modulo-2": "~1.0.0",
        "modulo-3": ">=1.0.0"
```

```
    },
    "devDependencies": {
        "modulo-4": "*"
    }
}
```

Com esses atributos, você já descreve o mínimo possível o que será sua aplicação. O atributo `name` é o principal, com ele você descreve o nome do projeto, nome pelo qual seu módulo será chamado via função `require('meu-primeiro-node-app')`. Em `description` descrevemos o que será este módulo. Ele deve ser escrito de forma curta e clara explicando um resumo do módulo. O `author` é um atributo para informar o nome e email do autor, utilize o formato: `Nome <email>` para que sites como (https://npmjs.org) reconheça corretamente esses dados. Outro atributo principal é o `version`, é com ele que definimos a versão atual do módulo, é extremamente recomendado que tenha este atributo, senão será impossível instalar o módulo via comando `npm`. O atributo `private` é um booleano, e determina se o projeto terá código aberto ou privado para download no (https://npmjs.org) .

Os módulos no Node.js trabalham com **3 níveis de versionamento**. Por exemplo, a versão `1.2.3` esta dividida nos níveis: *Major* (1), *Minor* (2) e *Patch* (3). Repare que no campo `dependencies` foram incluídos 4 módulos, cada módulo utilizou uma forma diferente de definir a versão que será adicionada no projeto. O primeiro, o `modulo-1` somente será incluído sua versão fixa, a `1.0.0`. Utilize este tipo versão para instalar dependências cuja suas atualizações possam quebrar o projeto pelo simples fato de que certas funcionalidades foram removidas e ainda as utilizamos na aplicação. O segundo módulo já possui uma certa flexibilidade de update. Ele utiliza o caractere ~ que faz atualizações a nível de *patch* `(1.0.x)`, geralmente essas atualizações são seguras, trazendo apenas melhorias ou correções de bugs. O `modulo-3` atualiza versões que seja maior ou igual a `1.0.0` em todos os níveis de versão. Em muitos casos utilizar ">=" pode ser perigoso, por que a dependência pode ser atualizada a nível *major* ou *minor*, contendo grandes modificações que podem quebrar um sistema em produção, comprometendo seu funcionamento e exigindo que você atualize todo código até voltar ao normal. O último, o `modulo-4`, utiliza o caractere "*", este sempre pegará a última versão do módulo em qualquer nível. Ele também pode causar problemas nas atualizações e tem o mesmo comportamento do versionamento do `modulo-3`. Geralmente ele é utilizado em `devDependencies`, que são dependências focadas para testes automatizados, e as atualizações dos módulos

não prejudicam o comportamento do sistema que já está no ar.

1.8 Escopos de variáveis globais

Assim como no browser, utilizamos o mesmo Javascript no Node.js, ele também utiliza **escopos locais e globais** de variáveis. A única diferença é como são implementados esses escopos. No client-side as variáveis globais são criadas da seguinte maneira:

```
window.hoje = new Date();
alert(window.hoje);
```

Em qualquer browser a palavra-chave `window` permite criar variáveis globais que são acessadas em qualquer lugar. Já no Node.js utilizamos uma outra *keyword* para aplicar essa mesma técnica:

```
global.hoje = new Date();
console.log(global.hoje);
```

Ao utilizar `global` mantemos uma variável global, acessível em qualquer parte do projeto sem a necessidade de chamá-la via `require` ou passá-la por parâmetro em uma função. Esse conceito de variável global é existente na maioria das linguagens de programação, assim como sua utilização, pelo qual é recomendado trabalhar com o mínimo possível de variáveis globais, para evitar futuros gargalos de memória na aplicação.

1.9 CommonJS, Como ele funciona?

O Node.js utiliza nativamente o padrão *CommonJS* para organização e carregamento de módulos. Na prática, diversas funções deste padrão será utilizada com frequência em um projeto Node.js. A função `require('nome-do-modulo')` é um exemplo disso, ela carrega um módulo. E para criar um código Javascript que seja modular e carregável pelo `require`, utilizam-se as variáveis globais: `exports` ou `module.exports`. Abaixo apresento-lhe dois exemplos de códigos que utilizam esse padrão do *CommonJS*, primeiro crie o código `hello.js`:

```
module.exports = function(msg) {
    console.log(msg);
};
```

E também crie o código `human.js` com o seguinte código:

```
exports.hello = function(msg) {
    console.log(msg);
};
```

A diferença entre o `hello.js` e o `human.js` esta na maneira de como eles serão carregados. Em `hello.js` carregamos uma única função modular e em `human.js` é carregado um objeto com funções modulares. Essa é a grande diferença entre eles. Para entender melhor na prática crie o código `app.js` para carregar esses módulos, seguindo o código abaixo:

```
var hello = require('./hello');
var human = require('./human');

hello('Olá pessoal!');
human.hello('Olá galera!');
```

Tenha certeza de que os códigos `hello.js`, `human.js` e `app.js` estejam na mesma pasta e rode no console o comando: `node app.js`.

E então, o que aconteceu? O resultado foi praticamente o mesmo, o `app.js` carregou os módulos: `hello.js` e `human.js` via `require()`, em seguida foi executado a função `hello()` que imprimiu a mensagem `Olá pessoal!` e por último o objeto `human` que executou sua função `human.hello('Olá galera!')`.

Percebam o quão simples é programar com Node.js! Com base nesses pequenos trechos de código já foi possível criar um código altamente escalável e modular que utiliza as boas práticas do padrão *CommonJS*.

Capítulo 2

Desenvolvendo aplicações web

2.1 Criando nossa primeira aplicação web

Node.js é multiprotocolo, ou seja, com ele será possível trabalhar com os protocolos: *HTTP, HTTPS, FTP, SSH, DNS, TCP, UDP, WebSockets* e também existem outros protocolos, que são disponíveis através de módulos não-oficiais criados pela comunidade. Um dos mais utilizados para desenvolver sistemas web é o protocolo *HTTP*. De fato, é o protocolo com a maior quantidade de módulos disponíveis para trabalhar no Node.js. Na prática desenvolveremos um sistema web utilizando o módulo nativo HTTP, mostrando suas vantagens e desvantagens. Também apresentarei soluções de módulos estruturados para desenvolver aplicações complexas de forma modular e escalável.

Toda aplicação web necessita de um servidor para disponibilizar todos os seus recursos. Na prática, com o Node.js você desenvolve uma *"aplicação middleware"*, ou seja, além de programar as funcionalidades da sua aplicação, você também programa códigos de configuração de infraestrutura da sua aplicação. Inicialmente isso parece ser muito trabalhoso, pois o Node.js utiliza o mínimo de configurações para servir

uma aplicação, mas esse trabalho permite que você customize ao máximo o seu servidor. Uma vantagem disso é poder configurar em detalhes o sistema, permitindo desenvolver algo performático e controlado pelo programador.

Caso performance não seja prioridade no desenvolvimento do seu sistema, recomendo que utilize alguns módulos adicionais que já vêm com o mínimo necessário de configurações prontas para você não perder tempo trabalhando com isso. Alguns módulos conhecidos são: **Connect (https://github.com/senchalabs/connect)** , **Express (http://expressjs.com)** , **Geddy (http://geddyjs.org)** , **CompoundJS (http://compoundjs.com)** , **Sails (http://balderdashy.github.io/sails)** . Esses módulos já são preparados para trabalhar desde uma infraestrutura mínima até uma mais enxuta, permitindo trabalhar desde arquiteturas *RESTFul*, padrão MVC *(Model-View-Controller)* e também com conexões real-time utilizando *WebSockets*.

Primeiro usaremos apenas o módulo nativo *HTTP*, pois precisamos entender todo o conceito desse módulo, visto que todos os frameworks citados acima o utilizam como estrutura inicial em seus projetos. Abaixo mostro a vocês uma clássica aplicação *Hello World*. Crie o arquivo `hello_server.js` com o seguinte conteúdo:

```
var http = require('http');

var server = http.createServer(function(request, response){
  response.writeHead(200, {"Content-Type": "text/html"});
  response.write("<h1>Hello World!</h1>");
  response.end();
});
server.listen(3000);
```

Esse é um exemplo clássico e simples de um servidor node.js. Ele está sendo executado na **porta 3000**, por padrão ele responde através da **rota raiz "/"** um resultado em formato html com a mensagem: `Hello World!`.

Vá para a linha de comando e rode `node hello_server.js`. Faça o teste acessando, no seu navegador, o endereço http://localhost:3000 .

2.2 Como funciona um servidor http?

Um servidor node.js utiliza o mecanismo *Event loop*, sendo responsável por lidar com a emissão de eventos. Na prática, a função `http.createServer()` é responsável por levantar um servidor e o seu callback `function(request, response)`

apenas é executado quando o servidor recebe uma requisição. Para isso, o *Event loop* constantemente verifica se o servidor foi requisitado e, quando ele recebe uma requisição, ele emite um evento para que seja executado o seu callback.

O Node.js trabalha muito com chamadas assíncronas que respondem através callbacks do javascript. Por exemplo, se quisermos notificar que o servidor está de pé, mudamos a linha `server.listen` para receber em parâmetro uma função que faz esse aviso:

```
server.listen(3000, function(){
  console.log('Servidor Hello World rodando!');
});
```

O método `listen` também é assíncrono e você só saberá que o servidor está de pé quando o Node invocar sua função de callback.

Se você ainda está começando com JavaScript, pode estranhar um pouco ficar passando como parâmetro uma `function` por todos os lados, mas isso é algo muito comum no mundo Javascript. Como sintaxe alternativa, caso o seu código fique muito complicado em encadeamentos de diversos blocos, podemos isolá-lo em funções com nomes mais significativos, por exemplo:

```
var http = require('http');

var atendeRequisicao = function(request, response) {
  response.writeHead(200, {"Content-Type": "text/html"});
  response.write("<h1>Hello World!</h1>");
  response.end();
}
var server = http.createServer(atendeRequisicao);

var servidorLigou = function() {
  console.log('Servidor Hello World rodando!');
}
server.listen(3000, servidorLigou);
```

2.3 Trabalhando com diversas rotas

Até agora respondemos apenas o endereço `/`, mas queremos possibilitar que nosso servidor também responda a outros endereços. Utilizando um palavreado comum entre desenvolvedores rails, queremos adicionar novas **rotas**.

Vamos adicionar duas novas rotas, uma rota `/bemvindo` para página de "Bem-vindo ao Node.js!" e uma rota genérica, que leva para uma página de erro. Faremos isso através de um simples encadeamento de condições, em um novo arquivo: `hello_server3.js`:

```
var http = require('http');
var server = http.createServer(function(request, response){
  response.writeHead(200, {"Content-Type": "text/html"});
  if(request.url == "/"){
    response.write("<h1>Página principal</h1>");
  }else if(request.url == "/bemvindo"){
    response.write("<h1>Bem-vindo :)</h1>");
  }else{
    response.write("<h1>Página não encontrada :(</h1>");
  }
  response.end();
});
server.listen(3000, function(){
  console.log('Servidor rodando!');
});
```

Rode novamente e faça o teste acessando a url http://localhost:3000/bemvindo , e também acessando uma outra, diferente desta. Viu o resultado?

Reparem na complexidade do nosso código: o roteamento foi tratado através dos comandos **if e else**, e a leitura de url é obtida através da função `request.url()` que retorna uma string sobre o que foi digitado na barra de endereço do browser. Esses endereços utilizam padrões para capturar valores na url. Esses padrões são: *query strings* (`?nome=joao`) e *path* (`/admin`). Em um projeto maior, tratar todas as urls dessa maneira seria trabalhoso e confuso demais. No Node.js, existe o módulo nativo chamado `url`, que é responsável por fazer *parser* e formatação de urls. Acompanhe como capturamos valores de uma query string no exemplo abaixo. Aproveite e crie o novo arquivo `url_server.js`:

```
var http = require('http');
var url = require('url');

var server = http.createServer(function(request, response){
  response.writeHead(200, {"Content-Type": "text/html"});
  response.write("<h1>Dados da query string</h1>");
  var result = url.parse(request.url, true);
```

```
  for(var key in result.query){
    response.write("<h2>"+key+" : "+result.query[key]+"</h2>");
  }
  response.end();
});
server.listen(3000, function(){
  console.log('Servidor http.');
});
```

Neste exemplo, a função **url.parse(request.url, true)** fez um parser da url obtida pela requisição do cliente `(request.url).`

Esse módulo identifica através do retorno da função **url.parser()** os seguintes atributos:

- **href:** Retorna a url completa: 'http://user:pass@host.com:8080/p/a/t/h?query=string#hash'
- **protocol:** Retorna o protocolo: 'http'
- **host:** Retorna o domínio com a porta: 'host.com:8080'
- **auth:** Retorna dados de autenticação: 'user:pass'
- **hostname:** Retorna o domínio: 'host.com'
- **port:** Retorna a porta: '8080'
- **pathname:** Retorna os pathnames da url: '/p/a/t/h'
- **search:** Retorna uma query string: '?query=string'
- **path:** Retorna a concatenação de pathname com query string: '/p/a/t/h?query=string'
- **query:** Retorna uma query string em JSON: {'query':'string'}
- **hash:** Retorna ancora da url: '#hash'

Resumindo, o módulo url permite organizar todas as urls da aplicação.

2.4 Separando o HTML do Javascript

Agora precisamos organizar os códigos HTML, e uma boa prática é separá-los do Javascript, fazendo com que a aplicação renderize código HTML quando o usuário solicitar uma determinada rota. Para isso, utilizaremos outro módulo nativo **FS** *(File System)*. Ele é responsável por manipular arquivos e diretórios do sistema operacional. O mais interessante desse módulo é que ele possui diversas funções de manipulação tanto de forma assíncrona como de forma síncrona. Por padrão, as funções nomeadas com o final `Sync()` são para tratamento síncrono. No exemplo abaixo, apresento as duas maneiras de ler um arquivo utilizando *File System*:

```
var fs = require('fs');
fs.readFile('/index.html', function(erro, arquivo){
  if (erro) throw erro;
  console.log(arquivo);
});
var arquivo = fs.readFileSync('/index.html');
console.log(arquivo);
```

Diversos módulos do Node.js possuem funções com versões assíncronas e síncronas. O `fs.readFile()` faz uma leitura assíncrona do arquivo `index.html`. Depois que o arquivo foi carregado, é invocado uma função callback para fazer os tratamentos finais, seja de erro ou de retorno do arquivo. Já o `fs.readFileSync()` realizou uma leitura síncrona, bloqueando a aplicação até terminar sua leitura e retornar o arquivo.

Limitações do File System nos sistemas operacionais

Um detalhe importante sobre o módulo *File System* é que ele não é 100% consistente entre os sistemas operacionais. Algumas funções são específicas para sistemas *Linux, OS X, Unix* e outras são apenas para *Windows*.

Para melhores informações leia sua documentação: http://nodejs.org/api/fs.html

Voltando ao desenvolvimento da nossa aplicação, utilizaremos a função `fs.readFile()` para renderizar html de forma assíncrona. Crie um novo arquivo, chamado `site_pessoal.js`, com o seguinte código:

```
var http = require('http');
var fs = require('fs');

var server = http.createServer(function(request, response){
  // A constante __dirname retorna o diretório raiz da aplicação.
  fs.readFile(__dirname + '/index.html', function(err, html){
    response.writeHeader(200, {'Content-Type': 'text/html'});
    response.write(html);
    response.end();
  });
});
server.listen(3000, function(){
  console.log('Executando Site Pessoal');
});
```

Para que isso funcione, você precisa do arquivo `index.html` dentro do mesmo diretório. Segue um exemplo de hello que pode ser utilizado:

```
<!DOCTYPE html>
<html>
   <head>
      <title>Olá este é o meu site pessoal!</title>
   </head>
   <body>
      <h1>Bem vindo ao meu site pessoal</h1>
   </body>
</html>
```

Rode o `node site_pessoal.js` e acesse novamente http://localhost:3000 .

2.5 Desafio: Implementar um roteador de url

Antes de finalizar esse capítulo, quero propor um desafio. Já que aprendemos a utilizar os **módulos http, url e fs (file system)**, que tal reorganizar a nossa aplicação para renderizar um determinado arquivo HTML baseado no path da url?

As regras do desafio são:

- Crie 3 arquivos HTML: `artigos.html, contato.html e erro.html`;
- Coloque qualquer conteúdo para cada página html;

- Ao digitar no browser o path: **/artigos** deve renderizar **artigos.html**;
- A regra anterior também se aplica para o arquivo **contato.html**;
- Ao digitar qualquer path diferente de **/artigos** e **/contato** deve renderizar **erro.html**;
- A leitura dos arquivos html deve ser assíncrona;
- A rota principal "/" deve renderizar **artigos.html**;

Algumas dicas importantes:

1) Utilize o retorno da função: `url.parse()` para capturar o **pathname** digitado e renderizar o html correspondente. Se o **pathname** estiver vazio significa que deve renderizar a página de artigos, e se estiver com um valor diferente do nome dos arquivos html, renderize a página de erros.
2) Você também pode inserir conteúdo html na função: `response.end(html)`, economizando linha de código ao não utilizar a função: `response.write(html)`.
3) Utilize a função: `fs.exists(html)` para verificar se existe o html com o mesmo nome do pathname digitado.

O resultado desse desafio se encontra na página github deste livro:
https://github.com/caio-ribeiro-pereira/livro-nodejs/tree/master/desafio-1

Capítulo 3

Por que o assíncrono?

3.1 Desenvolvendo de forma assíncrona

É importante focar no uso das chamadas assíncronas quando trabalhamos com Node.js, assim como entender quando elas são invocadas. O código abaixo exemplifica as diferenças entre uma função síncrona e assíncrona em relação a linha do tempo que ela são executadas. Basicamente criaremos um loop de 5 iterações, a cada iteração será criado um arquivo texto com o mesmo conteúdo `Hello Node.js!`. Primeiro vamos começar com o código síncrono. Crie o arquivo `text_sync.js` com o código abaixo:

```
var fs = require('fs');

for(var i = 1; i <= 5; i++) {
  var file = "sync-txt" + i + ".txt";
  var out = fs.writeFileSync(file, "Hello Node.js!");
  console.log(out);
}
```

Agora vamos criar o arquivo `text_async.js`, com seu respectivo código, diferente apenas na forma de chamar a função `writeFileSync`, que será a versão assíncrona `writeFile`, recebendo uma função como argumento:

```
var fs = require('fs');

for(var i = 1; i <= 5; i++) {
  var file = "async-txt" + i + ".txt";
  fs.writeFile(file, "Hello Node.js!", function(err, out) {
    console.log(out);
  });
}
```

Vamos rodar? Execute os comandos: `node text_sync` e depois `node text_async`. Se for gerado 10 arquivos no mesmo diretório do código-fonte, então deu tudo certo. Mas a execução de ambos foi tão rápida que não foi perceptível ver as diferenças entre o `text_async` e o `text_sync`. Para entender melhor as diferenças, veja as *timelines* que foram geradas. O `text_sync` por ser um código síncrono, invocou chamadas de I/O bloqueante, gerando o gráfico abaixo:

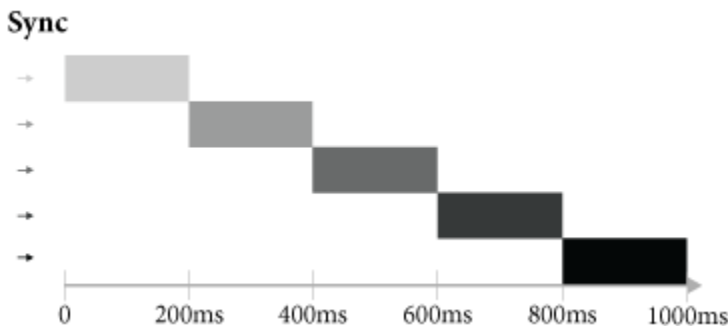



Figura 3.1: Timeline síncrona bloqueante.

Repare no tempo de execução, o `text_sync` demorou 1000 milissegundos, 200 milissegundos para cada arquivo criado.

Já em `text_async` foram criados os arquivos de forma totalmente assíncrona, ou seja, as chamadas de I/O eram não-bloqueantes, sendo executadas totalmente em paralelo:

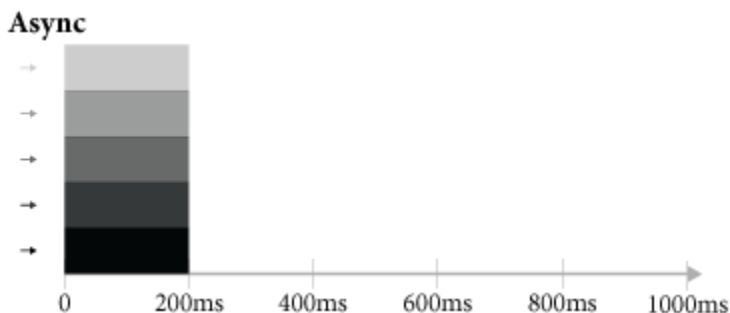



Figura 3.2: Timeline assíncrona não-bloqueante.

Isto fez com que o tempo de execução levasse 200 milissegundos, afinal foram invocados 5 vezes em paralelo a função `fs.writeFile()`, maximizando processamento e minimizando o tempo de execução.

Threads vs Assincronismos

Por mais que as funções assíncronas possam executar em paralelo várias tarefas, elas jamais serão consideradas uma Thread (por exemplo Threads do Java). A diferença é que Threads são manipuláveis pelo desenvolvedor, ou seja, você pode pausar a execução de uma Thread ou fazê-la esperar o término de uma outra. Chamadas assíncronas apenas invocam suas funções numa ordem de que você não tem controle, e você só sabe quando uma chamada terminou quando seu callback é executado.

Pode parecer vantajoso ter o controle sobre as Threads a favor de um sistema que executa tarefas em paralelo, mas pouco domínio sobre eles pode transformar seu sistema em um caos de travamentos *dead-locks*, afinal Threads são executadas de forma bloqueante. Este é o grande diferencial das chamadas assíncronas, elas executam em paralelo suas funções sem travar processamento das outras e principalmente sem bloquear o sistema principal.

É fundamental que o seu código Node.js invoque o mínimo possível de funções bloqueantes. Toda função síncrona impedirá, naquele instante, que o Node.js continue executando os demais códigos até que aquela função seja finalizada. Por

exemplo, se essa função fizer um *I/O* em disco, ele vai bloquear o sistema inteiro, deixando o processador ocioso enquanto ele utiliza outros recursos de hardware, como por exemplo leitura em disco, utilização da rede etc.

Sempre que puder, utilize funções assíncronas para aproveitar essa característica principal do Node.js.

Talvez você ainda não esteja convencido. A próxima seção vai lhe mostrar como e quando utilizar bibliotecas assíncronas não-bloqueantes, tudo isso através de teste prático.

3.2 Assincronismo versus Sincronismo

Para exemplificar melhor, os códigos abaixo representam um benchmark comparando o tempo de bloqueio de execução **assíncrona vs síncrona**. Para isso crie 3 arquivos: `processamento.js`, `leitura_async.js` e `leitura_sync.js`. Criaremos isoladamente o código `leitura_async.js` que faz leitura assíncrona:

```
var fs = require('fs');
var leituraAsync = function(arquivo){
  console.log("Fazendo leitura assíncrona");
  var inicio = new Date().getTime();
  fs.readFile(arquivo)
  var fim = new Date().getTime();
  console.log("Bloqueio assíncrono: "+(fim - inicio)+ "ms");
};
module.exports = leituraAsync;
```

Em seguida criaremos o código `leitura_sync.js` que faz leitura síncrona:

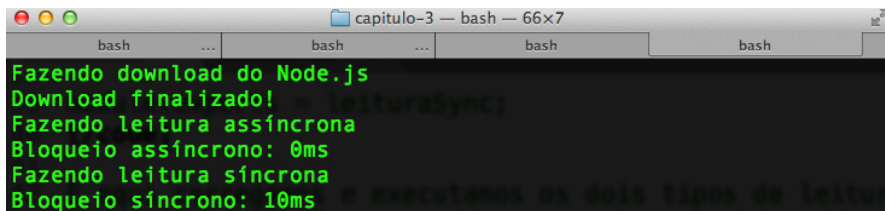
```
var fs = require('fs');
var leituraSync = function(arquivo){
  console.log("Fazendo leitura síncrona");
  var inicio = new Date().getTime();
  fs.readFileSync(arquivo);
  var fim = new Date().getTime();
  console.log("Bloqueio síncrono: "+(fim - inicio)+ "ms");
};
module.exports = leituraSync;
```

Para finalizar carregamos os dois tipos de leituras dentro do código `processamento.js`:

```
var http = require('http');
var fs = require('fs');
var leituraAsync = require('./leitura_async');
var leituraSync = require('./leitura_sync');
var arquivo = "./node.exe";
var stream = fs.createWriteStream(arquivo);
var download = "http://nodejs.org/dist/latest/node.exe";
http.get(download, function(res) {
  console.log("Fazendo download do Node.js");
  res.on('data', function(data){
    stream.write(data);
  });
  res.on('end', function(){
    stream.end();
    console.log("Download finalizado!");
    leituraAsync(arquivo);
    leituraSync(arquivo);
  });
});
```

Rode o comando `node processamento.js` para executar o benchmark. E agora, ficou clara a diferença entre o modelo bloqueante e o não-bloqueante?

Parece que o método `readFile` executou muito rápido, mas não quer dizer que o arquivo foi lido. Ele recebe um último parâmetro, que é um callback indicando quando o arquivo foi lido, que não passamos na invocação que fizemos.

Ao usar o `fs.readFileSync()`, bastaria fazer `var conteudo = fs.readFileSync()`. Mas qual é o problema dessa abordagem? **Ela segura todo o mecanismo do Node.JS**!

Basicamente este código fez o download de um arquivo grande (código-fonte do node.js) e, quando terminou, realizou um benchmark comparando o **tempo de bloqueio** entre as funções de leitura síncrona `(fs.readFileSync())` e assíncrona `(fs.readFile())` do Node.js.

Abaixo apresento o resultado do benchmark realizado em minha máquina:

- Modelo: MacBook Air 2011
- Processador: Core i5
- Memória: 4GB RAM

- Disco: 128GB SSD

Veja a pequena, porém significante diferença de tempo entre as duas funções de leitura.

Figura 3.3: Benchmark de leitura Async vs Sync.

Se esse teste foi com um arquivo de mais ou menos 50 MB, imagine esse teste em larga escala, lendo múltiplos arquivos de 1 GB ao mesmo tempo ou realizando múltiplos uploads em seu servidor? Esse é um dos pontos fortes do Node.js!

3.3 Entendendo o Event-Loop

Realmente trabalhar de forma assíncrona tem ótimos benefícios em relação a processamento I/O. Isso acontece devido ao fato, de que uma chamada de I/O é considerada um tarefa muito custosa para um computador realizar. Tão custosa que chega a ser perceptível para um usuário, por exemplo, quando ele tenta abrir um arquivo de 1 GB e o sistema operacional trava alguns segundos para abri-lo. Vendo o contexto de um servidor, por mais potente que seja seu *hardware*, eles terão os mesmos bloqueios perceptíveis pelo usuário, a diferença é que um servidor estará lidando com milhares usuários requisitando I/O, com a grande probabilidade de ser ao mesmo tempo.

É por isso que o Node.js trabalha com assincronismo. Ele permite que você desenvolva um sistema totalmente orientado a eventos, tudo isso graças ao *Event-loop*. Ele é um mecanismo interno, dependente das bibliotecas da linguagem C: `libev` (http://pod.tst.eu/http://cvs.schmorp.de/libev/ev.pod) e `libeio` (http://software.schmorp.de/pkg/libeio.html) , responsáveis por prover o assíncrono I/O no Node.js.

A ilustração abaixo apresenta como funciona o Event-Loop:

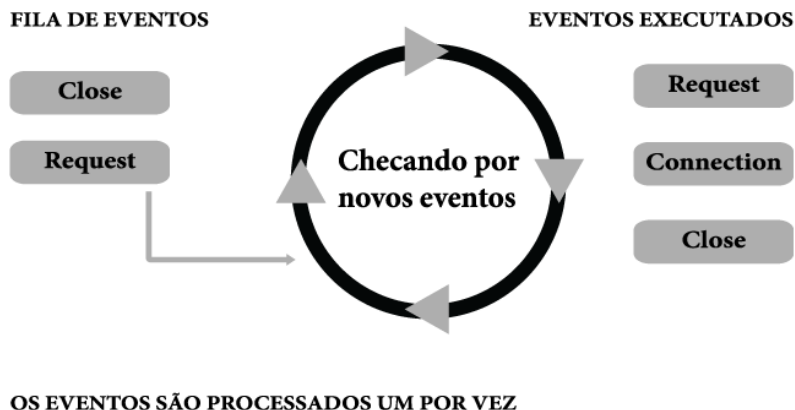



Figura 3.4: Event-Loop do Node.js.

Basicamente ele é um loop infinito, que em cada iteração verifica se existem novos eventos em sua **fila de eventos**. Tais eventos somente aparecem nesta fila quando são emitidos durante suas interações na aplicação. O `EventEmitter`, é o módulo responsável por emitir eventos e a maioria das bibliotecas do Node.js herdam deste módulo suas funcionalidades de eventos. Quando um determinado código emite um evento, o mesmo é enviado para a **fila de eventos** para que o *Event-loop* execute-o, e em seguida retorne seu *callback*. Tal callback pode ser executado através de uma função de escuta, semanticamente conhecida pelo nome: `on()`.

Programar orientado a eventos vai manter sua aplicação mais robusta e estruturada para lidar com eventos que são executados de forma assíncrona não-bloqueantes. Para conhecer mais sobre as funcionalidades do `EventEmitter` acesse sua documentação:

http://nodejs.org/api/events.html

3.4 Evitando Callbacks Hell

De fato, vimos o quanto é vantajoso e performático trabalhar de forma assíncrona, porém em certos momentos, inevitavelmente implementaremos diversas funções assíncronas, que serão encadeadas uma na outra através das suas funções callback. No código a seguir apresentarei um exemplo desse caso. Crie um arquivo chamado `callback_hell.js`, implemente e execute o código abaixo:

```
var fs = require('fs');
fs.readdir(__dirname, function(erro, contents) {
  if (erro) { throw erro; }
  contents.forEach(function(content) {
    var path = './' + content;
    fs.stat(path, function(erro, stat) {
      if (erro) { throw erro; }
      if (stat.isFile()) {
        console.log('%s %d bytes', content, stat.size);
      }
    });
  });
});
```

Reparem na quantidade de callbacks encadeados que existem em nosso código. Detalhe: ele apenas faz uma simples leitura dos arquivos de seu diretório e imprime na tela seu nome e tamanho em bytes. Um pequena tarefa como essa deveria ter menos encadeamentos, concorda? Agora, imagine como seria a organização disso para realizar tarefas mas complexas? Praticamente o seu código seria um caos e totalmente difícil de fazer manutenções. Por ser assíncrono, você perde o controle do que está executando em troca de ganhos com performance, porém, um detalhe importante sobre assincronismo é que na maioria dos casos os callbacks bem elaborados possuem como parâmetro uma variável de erro. Verifique nas documentações sobre sua existência e sempre faça o tratamento deles na execução do seu callback: `if (erro) { throw erro; }`, isso vai impedir a continuação da execução aleatória quando for identificado um erro.

Uma boa prática de código Javascript é criar funções que expressem seu objetivo e de forma isoladas, salvando em variável e passando-as como callback. Ao invés de criar funções anônimas, por exemplo, crie um arquivo chamado `callback_heaven.js` com o código abaixo:

```
var fs = require('fs');
var lerDiretorio = function() {
  fs.readdir(__dirname, function(erro, diretorio) {
    if (erro) return erro;
    diretorio.forEach(function(arquivo) {
      ler(arquivo);
    });
  });
};
var ler = function(arquivo) {
  var path = './' + arquivo;
  fs.stat(path, function(erro, stat) {
    if (erro) return erro;
    if (stat.isFile()) {
      console.log('%s %d bytes', arquivo, stat.size);
    }
  });
};
lerDiretorio();
```

Veja o quanto melhorou a legibilidade do seu código. Dessa forma deixamos mais semântico e legível o nome das funções e diminuímos o número de encadeamentos das funções de callback. A boa prática é ter o bom senso de manter no máximo até dois encadeamentos de callbacks. Ao passar disso significa que está na hora de criar uma função externa para ser passada como parâmetro nos callbacks, em vez de continuar criando um *callback hell* em seu código.

Capítulo 4

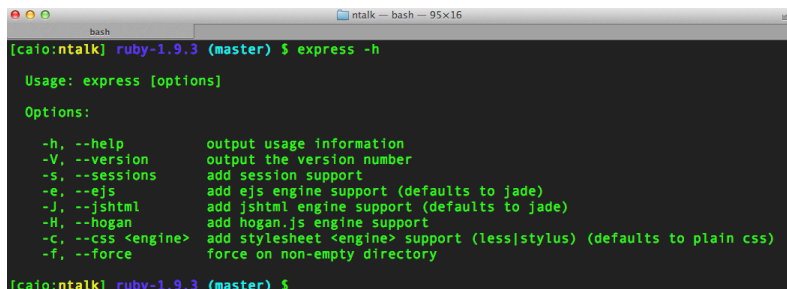
Iniciando com o Express

4.1 Por que utilizá-lo?

Programar utilizando apenas a API HTTP nativa é muito trabalhoso! Conforme surgem necessidades de implementar novas funcionalidades, códigos gigantescos seriam acrescentados, aumentando a complexidade do projeto e dificultando futuras manutenções.

Foi a partir desse problema que surgiu um framework muito popular, que se chama Express. Ele é um módulo para desenvolvimento de aplicações web de grande escala. Sua filosofia de trabalho foi inspirada pelo framework Sinatra da linguagem Ruby. O site oficial do projeto é: (http://expressjs.com) .

Figura 4.1: Framework Express.

Ele possui as seguintes características:

- MVR *(Model-View-Routes)*;
- MVC *(Model-View-Controller)*;
- Roteamento de urls via callbacks;
- Middleware;
- Interface *RESTFul*;
- Suporte a *File Uploads*;
- Configuração baseado em variáveis de ambiente;
- Suporte a *helpers* dinâmicos;
- Integração com *Template Engines*;
- Integração com SQL e NoSQL;

4.2 Instalação e configuração

Sua instalação é muito simples e há algumas opções de configurações para começar um projeto. Para aproveitar todos os seus recursos, recomendo que instale-o em modo global:

```
npm install -g express
```

Feito isso, será necessário fechar e abrir seu terminal para habilitar o comando `express`, que é um *CLI (Command Line Interface)* do framework. Ele permite gerar um projeto inicial com suporte a sessões, Template engine (por padrão ele inclui o framework Jade) e um CSS engine (por padrão ele utiliza CSS puro). Para visualizar todas as opções execute o comando: `express -h`

```
[caio:ntalk] ruby-1.9.3 (master) $ express -h

  Usage: express [options]

  Options:

    -h, --help          output usage information
    -V, --version       output the version number
    -s, --sessions      add session support
    -e, --ejs           add ejs engine support (defaults to jade)
    -J, --jshtml        add jshtml engine support (defaults to jade)
    -H, --hogan         add hogan.js engine support
    -c, --css <engine>  add stylesheet <engine> support (less|stylus) (defaults to plain css)
    -f, --force         force on non-empty directory

[caio:ntalk] ruby-1.9.3 (master) $
```

Figura 4.2: Opções do Express em modo CLI.

4.3 Criando um projeto de verdade

Vamos criar uma aplicação de verdade com Express? Dessa vez, criaremos um projeto que será trabalhado durante os demais capítulos do livro, e criaremos uma agenda de contatos em que seus contatos serão integrados em um web chat funcionando em real-time.

Os requisitos do projeto são:

- O usuário deve criar, editar ou excluir um contato;
- O usuário deve se logar informando seu nome e e-mail;
- O usuário deve conectar ou desconectar no chat;
- O usuário deve enviar e receber mensagens no chat somente entre os contatos online;

O nome do projeto será Ntalk *(Node talk)* e utilizaremos as seguintes tecnologias:

- **Node.js**: Backend do projeto;
- **MongoDB**: Banco de dados NoSQL orientado a documentos;

- **Redis**: Banco de dados NoSQL para estruturas de chave-valor;
- **Express**: Framework para aplicações web;
- **Socket.IO**: Módulo para comunicação real-time;
- **MongooseJS**: ODM *(Object Data Mapper)* MongoDB para Node.js;
- **Node Redis**: Cliente Redis para Node.js;
- **EJS**: *Template engine* para implementação de html dinâmico;
- **Mocha**: Framework para testes automatizados;
- **SuperTest**: Módulo para emular requisições que será utilizado no teste de integração;
- **Nginx**: Servidor Web de alta performance para arquivos estáticos;

Exploraremos estas tecnologias no decorrer desses capítulos, então muita calma e boa leitura!

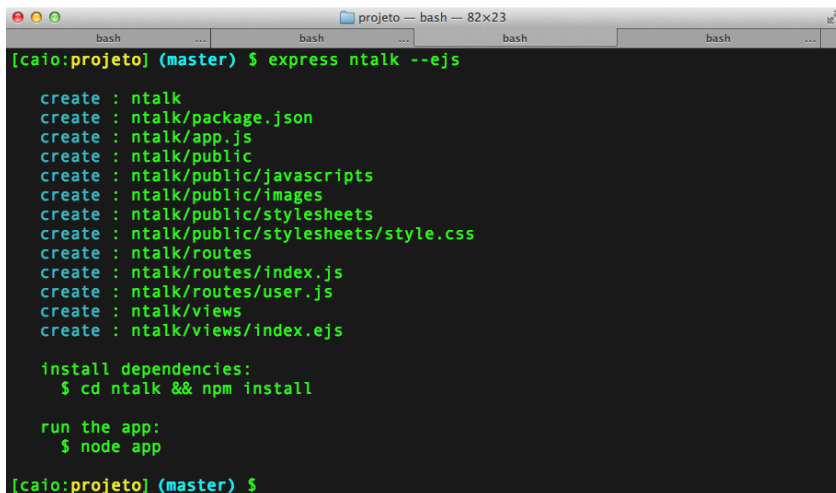
Caso você esteja com pressa de ver este projeto rodando, você pode cloná-lo através do meu repositório público: (https://github.com/caio-ribeiro-pereira/livro-nodejs) .

Para instalá-lo em sua máquina faça os comandos a seguir:

```
git clone git@github.com:caio-ribeiro-pereira/livro-nodejs.git
cd livro-nodejs/projeto/ntalk
npm install
npm start
```

E depois acesse no seu navegador favorito o endereço:

http://localhost:3000

Agora se você quer aprender passo a passo a desenvolver este projeto, continue lendo este livro, seguindo todas as dicas que irei passar.

Criaremos o diretório da aplicação já com alguns recursos do Express que é gerado a partir de seu CLI. Para começar, execute os seguintes comandos:

```
express ntalk --ejs
cd ntalk
npm install
```

Parabéns! Você acabou de criar o projeto **ntalk**.

4.4 Gerando scaffold do projeto

Ao acessar o diretório do projeto, veja como foi gerado o seu *scaffold*:

```
[caio:projeto] (master) $ express ntalk --ejs

   create : ntalk
   create : ntalk/package.json
   create : ntalk/app.js
   create : ntalk/public
   create : ntalk/public/javascripts
   create : ntalk/public/images
   create : ntalk/public/stylesheets
   create : ntalk/public/stylesheets/style.css
   create : ntalk/routes
   create : ntalk/routes/index.js
   create : ntalk/routes/user.js
   create : ntalk/views
   create : ntalk/views/index.ejs

   install dependencies:
     $ cd ntalk && npm install

   run the app:
     $ node app

[caio:projeto] (master) $
```

Figura 4.3: Estrutura do Express.

- `package.json`: contém as principais informações sobre a aplicação como: nome, autor, versão, colaboradores, url, dependências e muito mais.
- `public`: pasta pública que armazena conteúdo estático, por exemplo: imagens, css, javascript etc.
- `app.js`: arquivo que inicializa o servidor do projeto, através do comando: `node app.js`.
- `routes`: diretório que mantém todas as rotas da aplicação.
- `views`: diretório que contém todas as *views* que são renderizadas pelas rotas.

Ao rodarmos o comando `npm install`, por padrão ele instalou as dependências existentes no `package.json`. Neste caso, ele apenas instalou o Express e o EJS (Embedded Javascript).

Vamos fazer algumas alterações nos códigos gerados pelo *scaffold* do Express. O primeiro passo será criar uma descrição sobre o nosso projeto e definir `false` o atributo `private`. Isso tudo será modificado no arquivo `package.json`, veja o código abaixo como ficou:

```
{
  "name": "ntalk",
  "description": "Node talk - Agenda de contatos",
  "private": false,
  "version": "0.0.1",
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "3.4.7",
    "ejs": "0.8.5"
  }
}
```

Também modificaremos o `app.js`, deixando-o com o mínimo de código possível para explicarmos em *baby-steps* o que realmente será necessário no desenvolvimento deste projeto. Recomendo que apague o código gerado pelo *scaffold* e coloque o código abaixo:

```
var express = require('express')
  , routes = require('./routes');
var app = express();

app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
app.use(express.static(__dirname + '/public'));

app.get('/', routes.index);
app.get('/usuarios', routes.user.index);

app.listen(3000, function(){
  console.log("Ntalk no ar.");
});
```

Essa versão inicialmente atende os requisitos mínimos de uma aplicação Express. A brincadeira começa quando executamos a função `express()`, pois o seu retorno habilita todas as suas funcionalidades de seu framework, pelo qual armazenamos na variável `app`.

Com `app.listen()` fazemos algo parecido com o `http.listen()`, ou seja, ele é um *alias* responsável por colocar a aplicação no ar.

Os métodos `app.get()`, `app.post()`, `app.put()` e `app.del()` são funções de roteamento, cada uma delas associa seus respectivos métodos do protocolo HTTP *(GET, POST, PUT e DELETE)*. Seu primeiro parâmetro é uma string referente a uma rota da aplicação e o segundo é uma função callback que contém uma requisição e uma resposta. Exemplo: `app.get('/contatos', function(request, response));`

O `app.set(chave, valor)` é uma estrutura de chave e valor mantida dentro da variável `app`. Seria o mesmo que criar no javascript o código: `app["chave"] = "valor";`. Um exemplo prático são as configurações de `views` que foram definidos no código anterior: `(app.set('views', '/views'))` e `(app.set('view engine', 'ejs'))`.

A maioria das funções chamadas diretamente pela variável `express` são herdadas de seus submódulos `Connect` e `HTTP`.

Detalhes sobre Connect

O Connect (https://github.com/senchalabs/connect) é um middleware para servidores HTTP. Com ele é possível configurar aspectos do servidor através do conceito de pilha, ou seja, os primeiros itens inseridos são os primeiros a serem executados antes de chegar nas funções callbacks das rotas. O Express herdou todas as funcionalidades do Connect, por isso é recomendável compreender a ordem referente aos itens a serem incluídos no *stack* de configurações. Caso você não respeite a ordem de inserção de cada elemento, sua aplicação não irá se comportar bem, gerando erros inesperados ou não realizando tais rotinas que foram estabelecidas. Para isso, é necessário sempre dar uma olhada na documentação (http://www.senchalabs.org/connect) para saber mais sobre esses itens e também sobre a ordem de cada um. Um detalhe importante é que na própria documentação esses itens já estão listados na ordem em que cada um deve ser incluído na *stack* de sua aplicação.

Em nossa aplicação apenas inserimos dois itens na *stack* de configurações: o *template engine* EJS *(Embedded JavaScript)* e o diretório de arquivos estáticos. No decorrer deste livro serão incluídos e explicados novos itens no nosso projeto.

Também deixei apenas duas rotas: `/` e `/usuarios`, repare em como foram executados os seus callbacks; eles vieram da variável `var routes =`

`require('./routes')` pelo qual foi chamado um diretório e não um código javascript.

Por *default*, a chamada `routes.index` busca o arquivo `index.js` que contenha a função: `exports.index`, esta é uma convenção do Express para incluir a rota raiz da aplicação. Já o `routes.users.index` seguiu a regra normal de carregamento do arquivo `users.js` e sua respectiva função `exports.index`.

4.5 Organizando os diretórios do projeto

Quando o assunto é organização de códigos, o Express se comporta de forma bem flexível e liberal. Apesar de utilizar o seu *scaffold* de geração inicial, temos a total liberdade de modificar sua estrutura de diretórios e arquivos. Tudo vai depender da complexidade do projeto. Por exemplo, se o projeto for um sistema *single-page*, você pode desenvolver todo backend dentro código `app.js`, ou se o projeto possuir diversas rotas, *views*, *models* e *controllers*, o ideal seria montar uma estrutura modular utilizando o *pattern* MVC *(Model-View-Controller)*.

Em nosso projeto utilizaremos o padrão MVC, para isso falta criar os seguinte diretórios: `models` e `controllers`, deixando sua estrutura dessa forma:

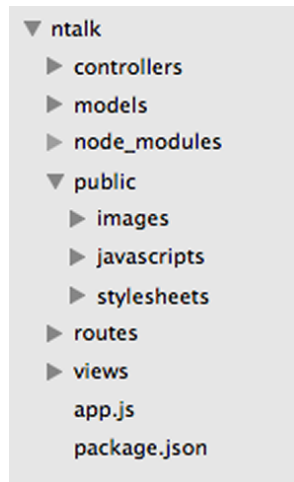



Figura 4.4: Estrutura de diretórios do ntalk.

Cada *model* que for utilizado em um *controller* realizará uma chamada a função `require('/models/nome-do-model');`. Em *controllers*, ou qualquer ou-

tro código, diversas chamadas a função `require` serão realizadas e isso vai gerar uma poluição nos códigos. O ideal é utilizarmos apenas para chamadas de módulos externos ou utilizar com frequência a função `require` dentro de `app.js`. Com base nesse problema, surgiu um *plugin* que visa minimizar essas chamadas, ele se chama **express-load**, sendo responsável por mapear diretórios para carregar e injetar módulos dentro de uma variável que definirmos na função `into`. Para entender melhor o que será feito, adicione-o como dependência no `package.json`:

```
"dependencies": {
  "express": "3.4.7",
  "express-load": "1.1.8",
  "ejs": "0.8.5"
}
```

Faça um update dos módulos para instalar o `express-load`, executando o comando: `npm install`.

Agora faremos um *refactoring* no `app.js` para implementarmos este módulo e utilizarmos sua função `load()`:

```
var express = require('express')
  , load = require('express-load')
  , app = express();

// ...stack de configurações do servidor...

load('models')
    .then('controllers')
    .then('routes')
    .into(app);

// ...app.listen(3000)...
```

É importante colocar em ordem os recursos a serem carregados pela função `load()`. Neste caso os *models* são carregados primeiro, em seguida vêm os *controllers*, e por último os *routes*.

Continuando o *refactoring*, exclua o arquivo `routes/user.js` que foi gerado pelo Express. E o código `routes/index.js`, vamos renomeá-lo para `routes/home.js`. Nele vamos incluir o seguinte código, que será uma adaptação para que ele utilize a variável `app` vinda do `express-load`:

```
module.exports = function(app) {
  var home = app.controllers.home;
  app.get('/', home.index);
};
```

O express-load criou um objeto chamado controllers dentro de app. Ele cria uma estrutura de objetos de acordo com o diretório e arquivo. Neste caso o app.controllers.home esta se referenciando ao arquivo controllers/home.js. Agora para esta rota funcionar corretamente, vamos criar o controller home.js e incluir sua primeira *action* chamada de index, seguindo o exemplo abaixo:

```
module.exports = function(app) {
  var HomeController = {
    index: function(req, res) {
      res.render('home/index');
    }
  };
  return HomeController;
};
```

Pronto! Para finalizar o fluxo entre routes e controllers, exclua o arquivo views/index.ejs e crie o diretório views/home. A nossa homepage será uma simples tela de login para acessar o sistema. Para fins ilustrativos a futura lógica desse aplicativo será de implementar um login que autocadastra um novo usuário, quando for informado um login novo no sistema. Em views/home crie o arquivo index.ejs, pelo qual implementaremos um formulário que conterá os campos nome e email. Veja como será esta *view* com base na implementação abaixo:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Ntalk - Agenda de contatos</title>
  </head>
  <body>
     <header>
        <h1>Ntalk</h1>
        <h4>Bem-vindo!</h4>
     </header>
     <section>
```

```
      <form action="/entrar" method="post">
        <input type="text" name="nome" placeholder="Seu nome">
        <br>
        <input type="text" name="email" placeholder="Seu e-mail">
        <br>
        <button type="submit">Entrar</button>
      </form>
    </section>
    <footer>
      <small>Ntalk - Agenda de contatos</small>
    </footer>
  </body>
</html>
```

Vamos rodar o projeto? Execute no terminal o comando: `node app.js`, em seguida acesse em seu *browser* o endereço: http://localhost:3000 Se tudo deu certo você verá uma tela com formulário semelhante a imagem abaixo:

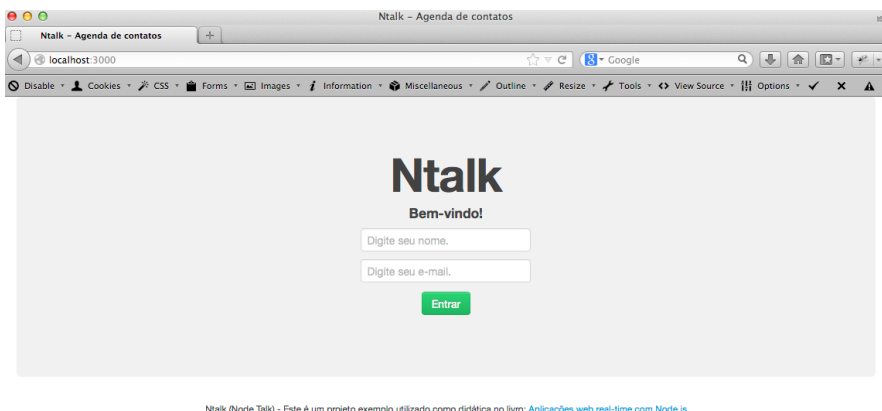



Figura 4.5: Tela de login do Ntalk.

Um detalhe a informar, nos exemplos deste livro não será implementado código css ou boas práticas de html, pois vamos focar apenas em código Javascript. Algumas imagens do projeto serão apresentados com um layout customizado. A versão completa deste projeto que inclui o código CSS e HTML de acordo com os *screenshots* pode ser acessada através do link do meu github:

http://github.com/caio-ribeiro-pereira/livro-nodejs/tree/master/projeto/ntalk

Parabéns! Acabamos de implementar o fluxo da tela inicial do projeto.

Capítulo 5

Dominando o Express

5.1 Estruturando views

O módulo EJS possui diversas funcionalidades que permitem programar conteúdo dinâmico em código html. Não entraremos a fundo neste framework, apenas utilizaremos seus principais recursos para renderizar conteúdo dinâmico e minimizar repetições de código. Com isso, isolaremos em outras *views*, conhecidas como *partials*, possíveis códigos que serão reutilizados com maior frequência. Dentro do diretório `views`, vamos criar dois arquivos que serão reaproveitados na homepage. O primeiro será o cabeçalho com o nome `header.ejs`:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Ntalk - Agenda de contatos</title>
  </head>
  <body>
```

E o segundo será o rodapé `footer.ejs`:

```
<footer>
      <small>Ntalk - Agenda de contatos</small>
    </footer>
  </body>
</html>
```

Agora modificaremos a nossa homepage, a `views/home/index.ejs` para que chamem esses *partials* através da função `include`:

```
<% include ../header %>
<header>
  <h1>Ntalk</h1>
  <h4>Bem-vindo!</h4>
</header>
<section>
  <form action="/entrar" method="post">
    <input type="text" name="usuario[nome]"
           placeholder="Digite o nome">
    <br>
    <input type="text" name="usuario[email]"
           placeholder="Digite o e-mail">
    <br>
    <button type="submit">Entrar</button>
  </form>
</section>
<% include ../footer %>
```

Agora a sua homepage ficou mais enxuta, fácil de ler e estruturada para reutilização de *partials*.

5.2 Controlando as sessões de usuários

Para o sistema fazer *login* e *logout* é necessário ter um controle de sessão. Esse controle permitirá que o usuário mantenha seus principais dados de acesso em memória no servidor, pois esses dados serão utilizados com maior frequência por grande parte do sistema. Trabalhar com sessão é muito simples, e os dados são manipulados através de objeto JSON dentro da variável: `req.session`.

Vamos aplicar um controle de sessão?. Primeiro, temos que criar duas novas rotas dentro de `routes/home.js`, sendo que uma será o `app.post('/entrar', home.login)` e a outra `app.get('/sair', home.logout)`:

```
module.exports = function(app) {
  var home = app.controllers.home;
  app.get('/', home.index);
  app.post('/entrar', home.login);
  app.get('/sair', home.logout);
};
```

Depois implementaremos suas respectivas *actions* no `controller/home.js`, seguindo a convenção de nomes `login` e `logout`, que foram utilizados no `routes/home.js`. Na *action login* será implementada uma simples regra de validação dos campos `nome` e `email` vindos do formulário. Se os campos passarem na validação, esses dados serão armazenados na sessão em `req.session.usuario`, assim como criarmos um array vazio (`usuario['contatos'] = [];`) de contatos para no futuro utilizá-lo na gestão de contatos. Em seguida será feito um redirecionamento para rota **/contatos**. Na *action logout* é chamada a função: `req.session.destroy()`, que irá limpar os dados da sessão e gerar uma nova.

```
module.exports = function(app) {
  var Usuario = app.models.usuario;

  var HomeController = {
    index: function(req, res) {
      res.render('home/index');
    },
    login: function(req, res) {
      var email = req.body.usuario.email
        , nome = req.body.usuario.nome;
      if(email && nome) {
        var usuario = req.body.usuario;
        usuario['contatos'] = [];
        req.session.usuario = usuario;
        res.redirect('/contatos');
      } else {
        res.redirect('/');
      }
    },
    logout: function(req, res) {
```

```
      req.session.destroy();
      res.redirect('/');
    }
  };
  return HomeController;
};
```

Vamos testar essa implementação? Dê um restart no servidor. Para isso finalize o servidor atual teclando no terminal `CTRL+C` (no Windows ou Linux) ou `Command+C` (no MacOSX), em seguida execute `node app.js` e depois em seu browser tente fazer um login no sistema.

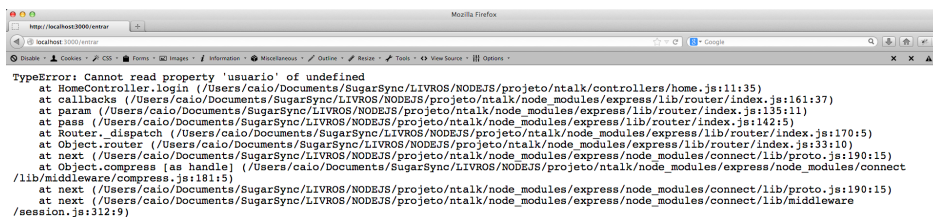



Figura 5.1: Infelizmente deu mensagem de erro.

O que houve? Eu implementei tudo certo! Então, meu amigo, esse erro aconteceu porque faltou habilitar um novo item na *stack* de configurações. Esse item é responsável por receber os dados do formulário html e fazer um *parser* para objeto JSON, afinal ele não reconheceu o objeto `req.body.usuario`. Já adiantando, também vamos incluir na *stack* um controle de sessão e cookies para que na próxima vez tudo funcione corretamente. Abaixo segue em ordem correta os novos itens da *stack* de configurações do nosso servidor:

```
app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
app.use(express.cookieParser('ntalk'));
app.use(express.session());
app.use(express.json());
app.use(express.urlencoded());
app.use(express.static(__dirname + '/public'));
```

É necessário incluir o `express.cookieParser()`, pois o `express.session()` utiliza-o para codificar e ou decodificar o `SessionID` que foi persistido no cookie. Outra configurações habilitadas foram o `express.json()` e `express.urlencoded()`, que são responsáveis por criar objetos JSON vindos de um formulário HTML. Ele cria um objeto através dos atributos name e value, existentes nas tags <input> , <select> e <textarea> . Ao submeter um formulário com a tag: <input name="usuario[idade] value="23"> será criado um objeto dentro de req.body . Neste caso, será criado req.body.usuario.idade

> **Alguns cuidados ao trabalhar com Sessions**
>
> Qualquer nome informado para o `req.session` será armazenado como um atributo deste objeto, por exemplo: `req.session.mensagem = "Olá".` Cuidado para não sobrescrever os nomes de suas funções nativas, como por exemplo: `req.session.destroy` ou `req.session.regenerate`. Ao fazer isso, você sobrescreve essas funções desabilitando suas respectivas funcionalidades. Com isso, no decorrer de sua aplicação possíveis inesperados bugs acontecerão no sistema. Para entender melhor as funções da `session`, veja a documentação do Connect:
>
> http://www.senchalabs.org/connect/session.html

Para finalizar, antes de testarmos as modificações, precisamos criar um `controller` e `routes` para contatos. Afinal temos a função `res.redirect('/contatos')` e até agora não foi implementada nenhuma lógica para realizar este redirecionamento.

A rota `/contatos` permite entrar em uma nova área do sistema, que será uma das principais áreas que iremos explorar no decorrer deste livro. Por enquanto vamos simplificar a implementação criando um *controller*, um rota e uma *view* para ele.

Crie o diretório `contatos` dentro de *views* e codifique o arquivo `index.ejs` com o seguinte conteúdo:

```
<% include ../header %>
<header>
```

```
  <h2>Ntalk - Agenda de contatos</h2>
</header>
<section>
  <p>Bem-vindo <%- usuario.nome %></p>
</section>
<% include ../exit %>
<% include ../footer %>
```

Nesta *view* vamos renderizar o nome do usuário logado, para isso utilizamos a função: `<%- usuario.nome %>`. Este objeto será enviado pelo controller que implementaremos a seguir. Repare que desta vez reaproveitamos o `header.ejs` e `footer.ejs` e isso tornou muito mais limpo esta *view*. Também incluímos um novo *partial* - `<% include ../exit %>`. Basicamente, ele possui o link **Sair**, que simplesmente faz *logout* no sistema. Ele será reaproveitado por grande parte do sistema, então iremos criá-lo dentro de `views/exit.ejs`:

```
<section>
  <a href='/sair'>Sair</a>
</section>
```

Agora vamos criar o seu controller chamado de `contatos.js`, contendo apenas a *action* `index`. Ele basicamente vai pegar os dados de um usuário logado através da variável `req.session.usuario` e vai enviá-los para *view* através da função: `res.render()`.

```
module.exports = function(app) {

  var ContatoController = {
    index: function(req, res) {
      var usuario = req.session.usuario
        , params = {usuario: usuario};
      res.render('contatos/index', params);
    }
  }
  return ContatoController;
};
```

Para finalizar, criaremos um novo *routes*, por convenção, utilizamos o mesmo nome do seu *controller* e por enquanto, vamos incluir a rota GET com o path: `/contatos`.

```
module.exports = function(app) {
  var contatos = app.controllers.contatos;
  app.get('/contatos', contatos.index);
};
```

Vamos testar novamente? Reinicie o servidor teclando no terminal `CTRL+C` (no Windows ou Linux) ou `Command+C` (no MacOSX), em seguida execute `node app.js`, por último acesse em seu browser: http://localhost:3000 . Faça novamente um login no sistema, dessa vez temos uma nova tela no sistema, a agenda de contatos.

Acabamos de expandir nosso projeto, incluímos a área principal. Aprendemos como habilitar session para implementar um controle de *login* e *logout*. Deixamos todas as views enxutas, isolando possíveis trechos de repetição de código html, além explorarmos novos itens da *stack* de configuração do servidor Node.

5.3 Criando rotas no padrão REST

A nossa agenda de contatos precisa ter como requisito mínimo um meio de permitir o usuário criar, listar, atualizar e excluir contatos. Esse é o conjunto clássico de funcionalidades, mais conhecido como CRUD *(Create, Receive, Update e Delete)*.

As rotas que utilizaremos para implementar o CRUD da agenda de contatos será aplicando o padrão de rotas REST. Esse padrão consiste em criar rotas utilizando os principais métodos do HTTP `(GET, POST, PUT e DELETE)`. Para isso teremos que habilitar um novo item na *stack* de configurações no `app.js`:

```
app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
app.use(express.cookieParser('ntalk'));
app.use(express.session());
app.use(express.json());
app.use(express.urlencoded());
app.use(express.methodOverride());
app.use(app.router);
app.use(express.static(__dirname + '/public'));
```

Incluímos o `express.methodOverride()` que permite utilizar um mesmo path entre os métodos do HTTP, fazendo uma sobrescrita de métodos. Também foi adicionado o *middleware* `app.router`, que gerencia as rotas da aplicação, permitindo a implementação de rotas para páginas de erros (muita calma! Isso ainda será

explicado neste capítulo.) e rotas para arquivos estáticos, sem conflitar com as rotas da aplicação.

Abra o arquivo `routes/contatos.js`, nele vamos implementar as futuras rotas para implementar o CRUD da agenda de contatos:

```
module.exports = function(app) {
  var contatos = app.controllers.contatos;
  app.get('/contatos', contatos.index);
  app.get('/contato/:id', contatos.show);
  app.post('/contato', contatos.create);
  app.get('/contato/:id/editar', contatos.edit);
  app.put('/contato/:id', contatos.update);
  app.del('/contato/:id', contatos.destroy);
};
```

Com as rotas criadas, precisamos agora implementar as regras de negócio dentro do controller `contatos.js`. Como ainda não utilizaremos um banco de dados, todos os dados serão persistidos na própria sessão do usuário, ou seja, todos os contatos serão gravados em memória e não em um banco de dados dedicado. Em `controller/contatos.js` implemente a seguinte lógica na *action* `index`:

```
module.exports = function(app) {
  var ContatoController = {
    index: function(req, res) {
      var usuario = req.session.usuario
        , contatos = usuario.contatos
        , params = {usuario: usuario
                  , contatos: contatos};
      res.render('contatos/index', params);
    },
    // continuação do controller...
}
```

Já na *action* `create` utilizaremos um simples *array* para persistir os contatos do usuário — `usuario.contatos.push(contato);` — para em seguida redirecionar o usuário para rota **/contatos**:

```
create: function(req, res) {
  var contato = req.body.contato
    , usuario = req.session.usuario;
  usuario.contatos.push(contato);
```

```
  res.redirect('/contatos');
},
// continuação do controller...
```

Em `show` e `edit` enviamos via parâmetro no path, o ID do usuário. Neste caso, passamos apenas o índice do contato referente a sua posição no array e em seguida enviamos o contato para a renderização de sua respectiva view `res.render('contatos/show')` e `res.render('contatos/edit');`:

```
show: function(req, res) {
  var id = req.params.id
    , contato = req.session.usuario.contatos[id]
    , params = {contato: contato, id: id};
  res.render('contatos/show', params);
},
edit: function(req, res) {
  var id = req.params.id
    , usuario = req.session.usuario
    , contato = usuario.contatos[id]
    , params = {usuario: usuario
              , contato: contato
              , id: id};
  res.render('contatos/edit', params);
},
// continuação do controller...
```

Agora temos a *actions* `update`. Ela recebe os dados de um contato atualizado que são submetidos pelo formulário da view `contatos/edit`. Em seguida utilizamos seu índice via `req.params.id` para atualizar no array o contato.

```
update: function(req, res) {
  var contato = req.body.contato
    , usuario = req.session.usuario;
  usuario.contatos[req.params.id] = contato;
  res.redirect('/contatos');
},
// continuação do controller...
```

Por último temos a *action* `destroy`, que basicamente recebe o índice através da variável `req.params.id` e exclui o contato do array via função `usuario.contatos.splice(id, 1)`.

```
destroy: function(req, res) {
    var usuario = req.session.usuario
      , id = req.params.id;
    usuario.contatos.splice(id, 1);
    res.redirect('/contatos');
  }
  // fim do controller...
  return ContatoController;
};
```

Esse foi um esboço muito básico da agenda de contatos. Porém, com esse esboço foi possível explorar algumas características do Express para a implementação de um CRUD.

Para finalizar, vamos criar as views para o usuário interagir no sistema. Dentro do diretório `views/contatos` vamos modificar a view `index.ejs` para renderizar uma lista de contatos e um formulário para cadastrar novos contatos:

```
<% include ../header %>
<header>
  <h2>Ntalk - Agenda de contatos</h2>
</header>
<section>
  <form action="/contato" method="post">
    <input type="text" name="contato[nome]" placeholder="Nome">
    <input type="text" name="contato[email]" placeholder="E-mail">
    <button type="submit">Cadastrar</button>
  </form>
  <table>
    <thead>
      <tr>
        <th>Nome</th>
        <th>E-mail</th>
        <th>Ação</th>
      </tr>
    </thead>
    <tbody>
      <% contatos.forEach(function(contato, index) { %>
        <tr>
          <td><%- contato.nome %></td>
          <td><%- contato.email %></td>
          <td>
```

```
            <a href="/contato/<%- index %>">Detalhes</a>
          </td>
        </tr>
      <% }) %>
    </tbody>
  </table>
</section>
<% include ../exit %>
<% include ../footer %>
```

Agora no mesmo diretório, vamos criar o `edit.ejs` e implementar um formulário para o usuário atualizar os dados de um contato:

```
<% include ../header %>
<header>
  <h2>Ntalk - Editar contato</h2>
</header>
<section>
  <form action="/contato/<%- id %>" method="post">
    <input type="hidden" name="_method" value="put">
    <label>Nome:</label>
    <input type="text" name="contato[nome]"
                       value="<%- contato.nome %>">
    <label>E-mail:</label>
    <input type="text" name="contato[email]"
                       value="<%- contato.email %>">
    <button type="submit">Atualizar</button>
  </form>
</section>
<% include ../exit %>
<% include ../footer %>
```

Por último e mais fácil de todos, crie a view `show.ejs`. Nela, vamos renderizar os dados de um contato que for selecionado. Incluiremos dois botões: **Editar** (para acessar a rota de edição do contato) e **Excluir** (botão que vai excluir o contato atual).

```
<% include ../header %>
<header>
  <h2>Ntalk - Dados do contato</h2>
</header>
<section>
  <form action="/contato/<%- id %>" method="post">
```

```
    <input type="hidden" name="_method" value="delete">
    <p><label>Nome:</label><%- contato.nome %></p>
    <p><label>E-mail:</label><%- contato.email %></p>
    <p>
      <button type="submit">Excluir</button>
      <a href="/contato/<%- id %>/editar">Editar</a>
    </p>
  </form>
</section>
<% include ../exit %>
<% include ../footer %>
```

Utilizando PUT e DELETE do HTTP

Infelizmente, as especificações atuais do HTTP não dão suporte para utilizar os verbos `PUT` e `DELETE` de forma semântica em um código html, como por exemplo:

```
<form action="/editar" method="put">
<form action="/excluir" method="delete">
```

A solução paliativa é criar a tag `<input>` com o nome `_method` e o valor `put` ou `delete`, para sobrescrever o método POST ou GET da tag `<form>`. Veja o exemplo abaixo:

```
<input type="hidden" name="_method" value="put">
<input type="hidden" name="_method" value="delete">
```

5.4 Aplicando filtros antes de acessar as rotas

Já percebeu que quando acessamos a rota `/contatos` sem logar no sistema acontece um bug? Não? Tente agora mesmo! Mas o que realmente provocou este erro? Bem, quando fazemos um login com sucesso, armazenamos os principais dados do usuário na sessão para utilizá-los no decorrer da aplicação. Quando acessamos `/contatos` sem antes ter feito um login, o controller tenta acessar a variável `req.session.usuario` que não existe ainda. Isso causa o seguinte bug na aplicação:

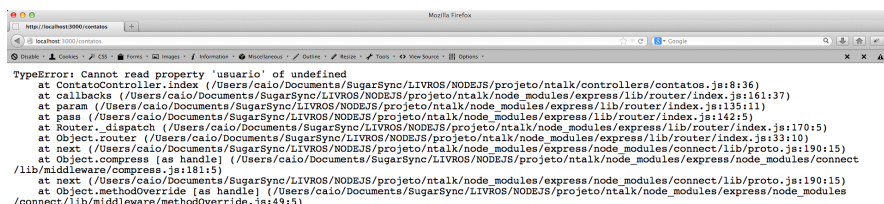



Figura 5.2: Bug: Cannot read property 'usuario' of undefined.

Diferente do Rails, Sinatra ou Django, o Express não possui filtros de forma explícita e em código legível. Não existe uma função `before` ou `after` pela qual se possa processar algo antes ou depois de entrar em uma rota. Mas calma! Nem tudo esta perdido! Graças ao Javascript temos as funções de callback, e o próprio mecanismo de roteamento do Express utiliza muito bem esse recurso, permitindo a criação de callback encadeados em uma rota. Resumindo, quando criamos uma rota, após informar no primeiro parâmetro a sua string, no segundo parâmetro em diante é possível incluir callbacks que são executados de forma ordenada, por exemplo: `app.get('/', callback1, callback2, callback3).`

Para implementarmos isso, primeiro vamos criar o filtro de autenticação. Ele será a criação do nosso primeiro *middleware* e será utilizado na maioria das rotas. Na raiz do projeto crie a pasta *middleware* incluindo o arquivo `autenticador.js`, nele implemente a lógica abaixo:

```
module.exports = function(req, res, next) {
  if(!req.session.usuario) {
    return res.redirect('/');
  }
  return next();
};
```

Esse filtro faz uma simples verificação se existe um usuário dentro da sessão. Se o usuário estiver autenticado, ou seja, estiver na `session`, será executado o callback `return next()` responsável por pular este filtro e indo para função ao lado. Caso autenticação não aconteça, executamos um simples `return`

res.redirect('/'), que faz o usuário voltar para página inicial e impedindo que ocorra o bug.

Com esse filtro implementado, agora temos que injetá-lo nos callbacks das rotas que precisam desse tratamento, faremos essas alterações no routes/contatos.js, de acordo com o código a seguir:

```
module.exports = function(app) {
  var autenticar = require('./../middleware/autenticador')
    , contatos = app.controllers.contatos;

  app.get('/contatos', autenticar, contatos.index);
  app.get('/contato/:id', autenticar, contatos.show);
  app.post('/contato', autenticar, contatos.create);
  app.get('/contato/:id/editar', autenticar, contatos.edit);
  app.put('/contato/:id', autenticar, contatos.update);
  app.del('/contato/:id', autenticar, contatos.destroy);
};
```

Basicamente inserimos o callback autenticar antes da função principal da rota. Isso nos permitiu emular a execução de um filtro before. Caso queira criar um filtro after, não há segredos: apenas coloque o callback do filtro por último. O importante é colocar as funções na ordem lógica de suas execuções.

5.5 Indo além: criando páginas de erros amigáveis

O Express oferece suporte para roteamento e renderização de erros do protocolo HTTP. Ele possui apenas duas funções, uma específica para tratamento do famoso erro **404** (página não encontrada) e uma função genérica que recebe por parâmetro uma variável contendo detalhes sobre o status e mensagem do erro HTTP.

Sobre o código de erros do HTTP

O protocolo HTTP tem diversos tipos de erros. O órgão W3C possui uma documentação explicando em detalhes o comportamento e código de cada erro. Para ficar por dentro desse assunto veja nesse link (http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html) a especificação dos status de erro gerados por este protocolo.

Que tal implementarmos um controle de erros para renderizar aos usuários páginas customizadas e mais amigáveis? Primeiro, crie duas novas views, uma para apresentar a tela do erro 404 conhecida na web como "**Página não encontrada**", seu nome será `not-found.ejs`...

```
<% include header %>
<header>
  <h1>Ntalk</h1>
  <h4>Infelizmente essa página não existe :(</h4>
</header>
<hr>
<p>Vamos voltar <a href="/">home page?</a> :)</p>
<% include footer %>
```

...e a outra será focada em mostrar erros gerados pelo protocolo HTTP com o nome de `server-error.ejs`:

```
<% include header %>
<header>
  <h1>Ntalk</h1>
  <h4>Aconteceu algo terrível! :(</h4>
</header>
<p>
  Veja os detalhes do erro:
  <br>
  <%- error.message %>
</p>
<hr>
<p>Que tal voltar <a href="/">home page?</a> :)</p>
<% include footer %>
```

Feito isso, agora só precisamos adicionar dois novos itens na *stack* de configurações do `app.js` para renderizar as respectivas views de erros:

```
app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
app.use(express.cookieParser('ntalk'));
app.use(express.session());
app.use(express.json());
app.use(express.urlencoded());
app.use(app.router);
app.use(express.static(__dirname + '/public'));
```

```
app.use(function(req, res, next) {
  res.status(404);
  res.render('not-found');
});
app.use(function(error, req, res, next) {
  res.status(500);
  res.render('server-error', error);
});
```

Para minimizar essa poluição de código na *stack* do servidor, exporte as funções de erros para um novo arquivo chamado `error.js` dentro do diretório *middleware*, deixando-as da seguinte forma:

```
exports.notFound = function(req, res, next) {
  res.status(404);
  res.render('not-found');
};
exports.serverError = function(error, req, res, next) {
  res.status(500);
  res.render('server-error', {error: error});
};
```

Em seguida modifique o *stack* do `app.js` para ficar mais limpo:

```
var express = require('express')
  , app = express()
  , load = require('express-load')
  , error = require('./middleware/error');

app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
app.use(express.cookieParser('ntalk'));
app.use(express.session());
app.use(express.json());
app.use(express.urlencoded());
app.use(app.router);
app.use(express.static(__dirname + '/public'));
app.use(error.notFound);
app.use(error.serverError);
```

Vamos testar o nosso código? Reinicie o servidor e, no browser, digite propositalmente uma nome de uma rota que não exista na aplicação, por exemplo:

http://localhost:3000/url-errada . Esta foi a renderização da tela de erro para página não encontrada (erro 404).

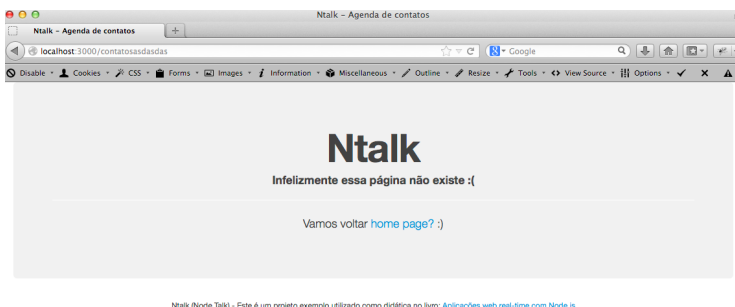



Figura 5.3: Tela de erro 404.

E para testar a página de erro interno do servidor? Esta tela somente será exibida em casos de erros graves no sistema. Para forçar um simples bug no sistema, remova um filtro da rota **/contatos**, forçando o bug que ocorria na seção anterior que falávamos sobre a implementação de filtros.

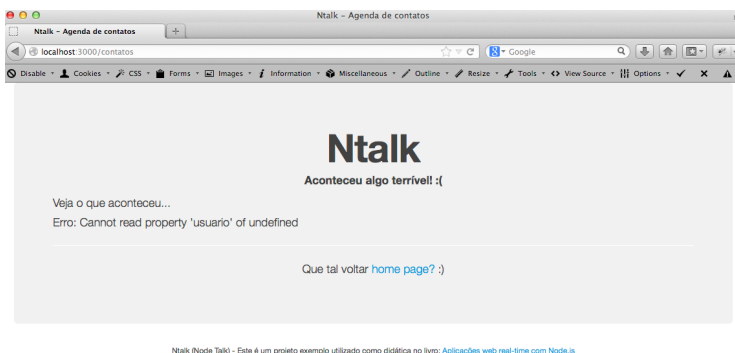



Figura 5.4: Tela de erro 500.

Parabéns! Agora temos uma aplicação que cadastra, edita, exclui e lista contatos. Utilizamos o padrão de rotas REST, criamos um simples controle de login que mantém o usuário na sessão, implementamos filtros para barrar acesso de usuários não autenticados e pra finalizar, implementamos a renderização de páginas de er-

ros amigáveis — afinal erros inesperados podem ocorrer em nossa aplicação e seria desagradável o usuário ver informações complexas na tela.

Capítulo 6

Programando sistemas real-time

6.1 Como funciona uma conexão bidirecional?

Este capítulo será muito interessante, pois falaremos sobre um assunto emergente nos sistemas atuais que está sendo largamente utilizado no Node.js. Estou falando sobre desenvolvimento de aplicações real-time.

Tecnicamente, estamos falando de uma conexão bidirecional, que, na prática, é uma conexão que se mantém aberta *(connection keep-alive)* para clientes e servidores interagirem em uma única conexão. A vantagem disso fica para os usuários, pois a interação no sistema será em tempo real, trazendo uma experiência de usuário muito melhor.

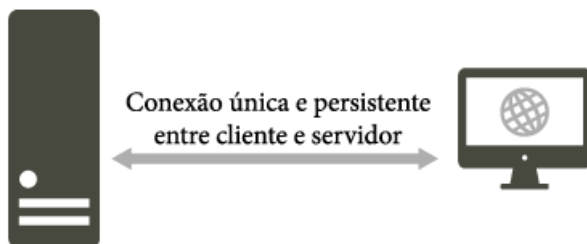



Figura 6.1: Imagem explicando sobre conexão bidirecional.

O Node.js se tornou popular por oferecer bibliotecas de baixo nível que suportam diversos protocolos *(HTTP, HTTPS, FTP, DNS, TCP, UDP e outros)*. O recente protocolo *WebSockets* também é compatível com Node.js e ele permite desenvolver sistemas de conexão persistente utilizando Javascript tanto no cliente quanto no servidor.

Infelizmente o único problema em utilizar este protocolo, é que nem todos os browsers suportam esse recurso, tornando inviável desenvolver uma aplicação real-time *cross-browser*.

6.2 Conhecendo o framework Socket.IO

Diante desse problema, nasceu o Socket.IO. Ele resolve a incompatibilidade entre o *WebSockets* com os navegadores antigos, emulando, por exemplo, *Ajax long-polling* ou outros *transports* de comunicação em browsers que não possuem *WebSockets*, de forma totalmente abstraída para o desenvolvedor. Seu site oficial é: (http://socket.io)

Figura 6.2: Framework Socket.IO.

O Socket.IO funciona da seguinte maneira: é incluído um script no cliente que detecta informações sobre o browser do usuário para definir qual será a melhor comunicação com o servidor. Os *transports* de comunicação que ele executa são:

1) *WebSocket*;

2) *Adobe Flash Socket*;

3) *AJAX long polling*;

4) *AJAX multipart streaming*;

5) *Forever iframe*;

6) *JSONP Polling*;

Se o navegador do usuário possuir compatibilidade com *WebSockets* ou *FlashSockets* (utilizando *Adobe Flash Player* do navegador), será realizada uma comunicação bidirecional. Caso contrário, será emulada uma comunicação unidirecional, que em curtos intervalos de tempo faz requisições AJAX no servidor. É claro que o desempenho é inferior, porém garante compatibilidade com browsers antigos e mantém o mínimo de experiência real-time para o usuário. O mais interessante de tudo isso é que programar utilizando o Socket.IO é muito simples e toda decisão complexa é ele que faz, simplificando a vida o desenvolvedor.

6.3 Implementando um chat real-time

Vamos ver como funciona na prática? Criaremos o web chat no ntalk, com o qual o usuário enviará mensagens para os usuários online da agenda de contatos. Integraremos o frameworks: Socket.IO no Express. Primeiro, instalaremos o módulo atualizando o `package.json`:

```
"dependencies": {
  "express": "3.4.7",
  "express-load": "1.1.8",
  "ejs": "0.8.5",
  "socket.io": "0.9.16"
}
```

Utilizaremos a versão `0.9.16` do Socket.IO, agora vamos instalá-lo executando no console: `npm install`

Vamos adaptar o `app.js` com o novo módulo. A função *listen* do servidor web será realizada via módulo http através da função `server.listen(3000)` para que o Socket.IO utilize-a para criar seu ponto de comunicação através do protocolo HTTP.

```
var express = require('express')
  , app = express()
  , load = require('express-load')
  , error = require('./middleware/error')
  , server = require('http').createServer(app)
  , io = require('socket.io').listen(server);

// ...código do stack de configurações...
// ...código da função load()...

io.sockets.on('connection', function (client) {
  client.on('send-server', function (data) {
    var msg = "<b>"+data.nome+":</b> "+data.msg+"<br>";
    client.emit('send-client', msg);
    client.broadcast.emit('send-client', msg);
  });
});

server.listen(3000, function(){
  console.log("Ntalk no ar.");
});
```

Instanciamos os módulos: `express`, `http` e `socket.io`. É necessário seguir esta ordem, pois o Socket.IO funciona a partir de uma instância do módulo http. Toda brincadeira começa a partir do evento: `io.sockets.on('connection')`. Esta função espera um cliente interagir com um de seus eventos internos. Por enquanto, o único evento interno é o `client.on('send-server')`, cuja sua execução ocorre quando um cliente envia uma mensagem para o servidor. Perceba que pouco código foi incluído e o servidor responde a seus clientes através das funções `client.emit('send-client')` e `client.broadcast.emit('send-client')`. Resumindo, o fluxo básico é o cliente enviar uma mensagem para o servidor e o servidor responde o próprio cliente (via `client.emit()`) e seus demais clientes conectados (via

`client.broadcast.emit()`). Para compreender melhor, veja as ilustrações abaixo:

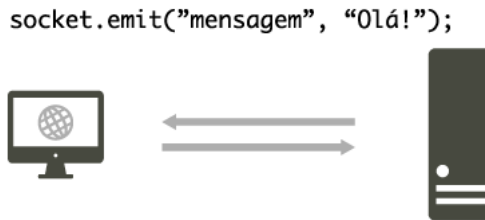



Figura 6.3: Envia mensagens para o cliente ou servidor.

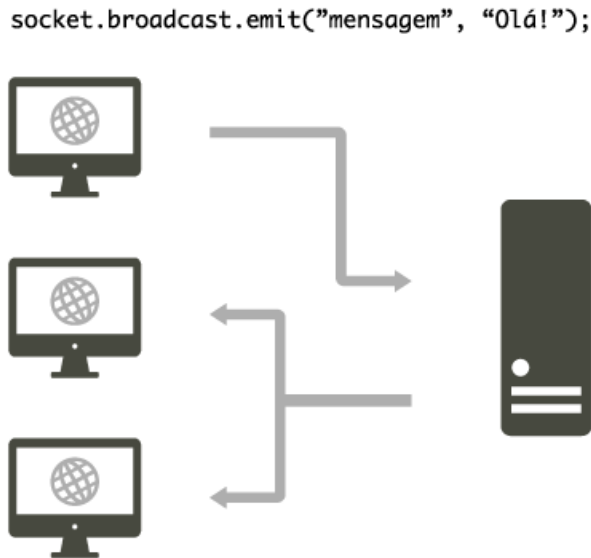



Figura 6.4: Envia mensagens para todos os clientes, exceto o próprio emissor.

Com o back-end do Socket.IO implementado, vamos modificar o front-end para interagir neste meio de comunicação. Dentro de `views/contatos/index.ejs` modificaremos a listagem dos contatos incluindo um link "**Conversar**", ele terá o path: **/chat/email-do-contato**, que será usado para criar a sala do chat respectiva para o seu contato.

```
<% contatos.forEach(function(contato, index) { %>
  <tr>
    <td><%- contato.nome %></td>
    <td><%- contato.email %></td>
    <td>
      <a href="/contato/<%- index %>">Detalhes</a>
      <a href="/chat/<%- contato.email %>">Conversar</a>
    </td>
  </tr>
<% }) %>
```

Vamos implementar o layout do nosso chat e principalmente os eventos de comunicação do cliente para interagir com o servidor. Crie um novo controller em `controller/chat.js`:

```
module.exports = function(app) {
  var ChatController = {
    index: function(req, res){
      var resultado = {email: req.params.email,
                       usuario: req.session.usuario};
      res.render('chat/index', resultado);
    }
  };
  return ChatController;
};
```

Crie o seu respectivo routes `routes/chat.js`:

```
module.exports = function(app) {
  var chat = app.controllers.chat;
  app.get('/chat/:email', chat.index);
};
```

E, por último, crie sua view em `views/chat/index.ejs`. Utilizaremos javascript puro para manipulação do html, mas sinta-se à vontade para utilizar qualquer framework do gênero, como por exemplo: *jQuery* (http://jquery.com) ou *ZeptoJS* (http://zeptojs.com) .

O importante é carregar o script `/socket.io/socket.io.js` para que a brincadeira comece. Aliás, não se preocupe de onde veio esse script, pois ele é distribuído automaticamente pelo framework `socket.io-client` que já vem como

dependência do Socket.IO, e ele é responsável pela comunicação do cliente com o servidor.

É pela função `io.connect` que o cliente se conecta com o servidor e começa a trocar mensagens com ele. Essa interação ocorre através do tráfego de objetos JSON. Vamos incluir no `index.ejs` as funções de enviar e receber mensagens:

```
<% include ../header %>
<script src="/socket.io/socket.io.js"></script>
<script>
  var socket = io.connect('http://localhost:3000');
  socket.on('send-client', function (msg) {
    document.getElementById('chat').innerHTML += msg;
  });
  var enviar = function() {
    var nome = document.getElementById('nome').value;
    var msg = document.getElementById('msg').value;
    socket.emit('send-server', {nome: nome, msg: msg});
  };
</script>
<header>
  <h2>Ntalk - Chat</h2>
</header>
<section>
  <pre id="chat"></pre>
  <input type="hidden" id="nome" value="<%- usuario.nome %>">
  <input type="text" id="msg" placeholder="Mensagem">
  <button onclick="enviar();">Enviar</button>
</section>
<% include ../exit %>
<% include ../footer %>
```

Em seguida, vamos reiniciar a aplicação com o comando `node app.js` e depois acessá-lo no endereço http://localhost:3000 pelo browser.

Para entender melhor como tudo funciona, acesse o mesmo endereço em outro browser para ter duas janelas na mesma aplicação. Cadastre dois usuários, cada um em sua respectiva janela, em seguida adicione em contatos o nome e e-mail do outro usuário, por exemplo, na janela do **Usuário A** adicione como contato os dados do **Usuário B** e vice-versa. Agora clique em no link **"Conversar"** e envie mensagem para o usuário.

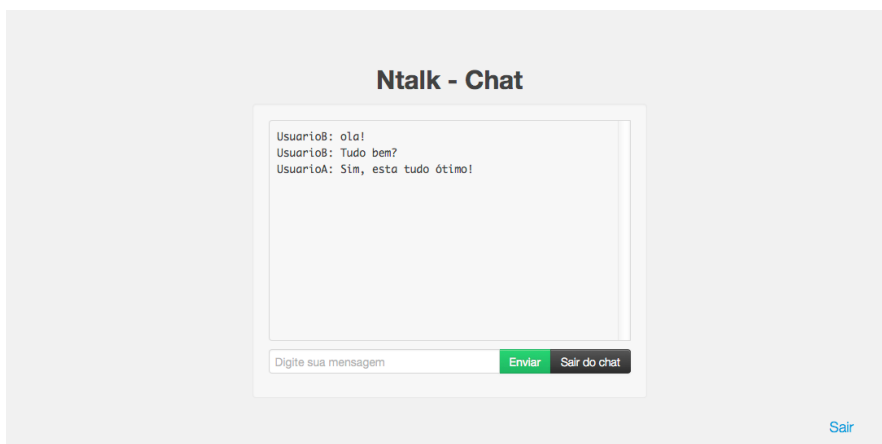



Figura 6.5: Usuário A conversando com Usuário B.

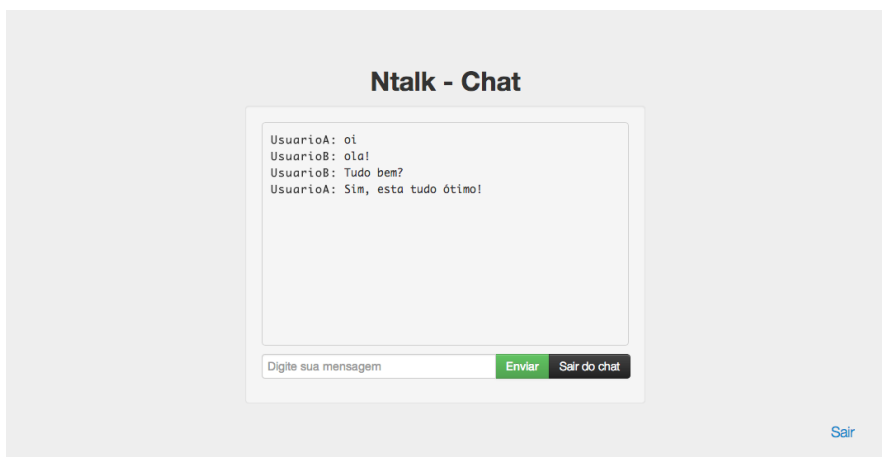



Figura 6.6: Usuário B respondendo mensagens do Usuário A.

Se tudo deu certo, parabéns! O web chat com o mínimo de recursos funcionais e em tempo real esta no ar. Mesmo que você use um navegador antigo, como o Internet Explorer 6, tudo funcionará bem! Pois o Socket.IO fará o trabalho de emular a comunicação entre ambos, mesmo sem a presença do protocolo WebSocket.

6.4 Organizando o carregamento de Sockets

A nossa aplicação terá duas funcionalidades com resposta em tempo real: o chat e o notificador de mensagens. Em sistemas maiores podem surgir diversas funcionalidades nessa modalidade e codificar todas as funções do Socket.IO em um único arquivo ficaria terrivelmente assustador para fazer manutenções. O ideal é separar as funcionalidades de forma modular e escalável — essa prática é semelhante ao que fizemos com as views, models e controllers. Com base nesse problema, crie o diretório `sockets` e adicione nele o arquivo `chat.js`. Para esse arquivo iremos migrar o código Socket.IO do `app.js`, veja o exemplo abaixo:

```
module.exports = function(io) {
  var sockets = io.sockets;
  sockets.on('connection', function (client) {
    client.on('send-server', function (data) {
      var msg = "<b>"+data.nome+":</b> "+data.msg+"<br>";
      client.emit('send-client', msg);
      client.broadcast.emit('send-client', msg);
    });
  });
}
```

Dessa forma deixamos o `app.js` com menos código. Um detalhe importante é que o `chat` será carregado por uma nova chamada a função `load()` pela qual passaremos como parâmetro a variável `io`:

```
var express = require('express')
  , app = express()
  , load = require('express-load')
  , error = require('./middleware/error')
  , server = require('http').createServer(app)
  , io = require('socket.io').listen(server)
;
// stack de configurações...
load('models')
  .then('controllers')
  .then('routes')
  .into(app);
load('sockets')
  .into(io);
// server.listen()...
```

E mais uma vez utilizamos boas práticas de código modular, organizando melhor o projeto e preparando-o para trabalhar com diversas funções do Socket.IO em conjunto com Express.

6.5 Socket.IO e Express em uma mesma sessão

O Socket.IO consegue acessar e manipular uma session criada pelo servidor web. Implementaremos esse controle de session compartilhada, pois é mais seguro do que passar os dados do usuário logado através da tag:

```
<input type="hidden" id="nome" value="<%- usuario.nome %>">
```

Como isso funciona? Na prática, quando logamos no sistema, o Express cria um ID de session para o usuário. Essa session é persistida em memória ou disco no servidor (essa decisão fica a critério dos desenvolvedores). O Socket.IO não consegue acessar esses dados, ele apenas possui um controle para autorizar uma conexão do cliente. Com isso, podemos utilizar as funções de session e cookies do Express dentro dessa função de autorização buscando e validando uma session — se o mesmo for válido, armazenamos no cliente Socket.IO, autorizando sua conexão no sistema.

Resumindo, precisamos criar um controle para compartilhar session entre o Express e Socket.IO. Vamos configurar no Express para isolar em variáveis as funções: `express.cookieParser` e `express.session`. Também criaremos duas constantes chamadas: `KEY` e `SECRET` que serão utilizadas para buscar o ID da session e carregar os dados do usuário logado utilizando o objeto `MemoryStore`. Faremos essas modificações no `app.js`, seguindo o trecho do código abaixo:

```
// carregamento dos módulos...
const KEY = 'ntalk.sid', SECRET = 'ntalk';
var cookie = express.cookieParser(SECRET)
  , store = new express.session.MemoryStore()
  , sessOpts = {secret: SECRET, key: KEY, store: store}
  , session = express.session(sessOpts);

app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
app.use(cookie);
app.use(session);
app.use(express.json());
app.use(express.urlencoded());
```

```
app.use(express.methodOverride());
app.use(app.router);
app.use(express.static(__dirname + '/public'));
app.use(error.notFound);
app.use(error.serverError);
```

Com esses recursos habilitados, será possível utilizar session no Socket.IO. Vamos implementar um controle de autorização via `io.set('authorization')` para que a cada conexão o Socket.IO valide o SessionID permitindo ou não recuperar os dados do usuário presente no sistema. A seguir, implementamos diversas condicionais para esse validação:

```
// require dos módulos...
// stack de configurações...
io.set('authorization', function(data, accept) {
  cookie(data, {}, function(err) {
    var sessionID = data.signedCookies[KEY];
    store.get(sessionID, function(err, session) {
      if (err || !session) {
        accept(null, false);
      } else {
        data.session = session;
        accept(null, true);
      }
    });
  });
});
// load()...
// server.listen()...
```

A função `accept()` é a responsável pela autorização da conexão e a variável `data` contém informações do cliente, isso inclui headers, cookies e outras informações do HTTP. Buscamos o `sessionID` através da variável `data.signedCookies[KEY]`, em seguida buscamos os dados da session que estão na memória do servidor através da função `store.get()`. Se tudo ocorrer com sucesso, incluímos a session na variável `data` e liberamos a conexão pela função `accept(null, true)`.

Pronto! Agora o Socket.IO esta habilitado para ler e manipular os objetos de uma session criada pelo Express. Com isso, podemos trafegar os dados do usuário

logado dentro do nosso chat. Para finalizar essa tarefa, faremos alguns *refactors*. Primeiro vamos deixar mais enxuta a view: `chat/index.ejs` removendo as variáveis referentes ao nome do usuário:

```
<% include ../header %>
<script src="/socket.io/socket.io.js"></script>
<script>
  var socket = io.connect('http://localhost:3000');
  socket.on('send-client', function (msg) {
    var chat = document.getElementById('chat');
    chat.innerHTML += msg;
  });
  var enviar = function() {
    var msg = document.getElementById('msg');
    socket.emit('send-server', msg.value);
  };
</script>
<header>
  <h2>Ntalk - Chat</h2>
</header>
<section>
  <pre id="chat"></pre>
  <input type="text" id="msg" placeholder="Mensagem">
  <input type="button" onclick="enviar();" value="Enviar">
</section>
<% include ../exit %>
<% include ../footer %>
```

Também vamos remover do controller `chat.js` a variável referente aos dados usuário da session `req.session.usuario`:

```
module.exports = function(app) {
  var ChatController = {
    index: function(req, res){
      var params = {email: req.params.email};
      res.render('chat/index', params);
    }
  };
  return ChatController;
};
```

Com a view e o controller mais limpa, vamos adaptar no `sockets/chat.js` o evento `sockets.on('connection')` para concatenar a string de mensagens com o nome do usuário. A única diferença é que agora serão carregados pela variável `client.handshake.session` os dados do usuário conectado:

```
module.exports = function(io) {
  var sockets = io.sockets;
  sockets.on('connection', function (client) {
    var session = client.handshake.session
      , usuario = session.usuario;
    client.on('send-server', function (msg) {
      msg = "<b>"+usuario.nome+":</b> "+msg+"<br>";
      client.emit('send-client', msg);
      client.broadcast.emit('send-client', msg);
    });
  });
}
```

6.6 Gerenciando salas do chat

Para finalizar o nosso chat, vamos aprimorá-lo implementando um controle de sala *one-to-one* para assegurar que cada conversa seja entre dois usuários no nosso bate-papo, prevenindo que outros usuários entrem no meio de uma conversa a dois.

Desenvolver essa funcionalidade é muito simples, apenas temos que criar uma string que será o nome da sala e utilizá-la através da função: `sockets.in('nome_da_sala').emit()`. Outro detalhe dessa função é que ela emite um evento para todos os usuários da sala, incluindo o próprio usuário emissor.

Para implementar uma sala *one-to-one* temos que garantir que em uma sala entrem no máximo dois clientes. Para implementar de forma segura temos que criar nomes de salas difíceis de serem decifrados, para isso utilizaremos um módulo nativo do Node.js para criptografar o nome de uma sala. Ele será um *Hash MD5* de um *timestamp* criado pelo primeiro usuário que começar uma conversa.

Dicas sobre criptografia no Node.js

A cada nova versão da API Crypto do Node.js é aprimorado novas melhorias e novas funções. Porém essa API trabalha com recursos básicos de criptografia, caso você seja um desenvolvedor focado em segurança, utilize o módulo BCrypt, que possui uma série de funcionalidades e algoritmos de criptografia mais difíceis de se quebrar. Para conhecer o BCrypt em detalhes e suas funcionalidades acesse sua documentação:

https://github.com/ncbooogt/node.bcrypt.js

Veja na prática como gerenciaremos as salas dos usuários. Primeiro carregaremos os módulos de criptografia para gerar o valor *hash* da sala.

```
module.exports = function(io) {
  var crypto = require('crypto')
    , md5 = crypto.createHash('md5')
    , sockets = io.sockets;
  // ...continuação do código...
}
```

Dentro do evento `sockets.on('connection')`, criaremos um novo evento `client.on('join')`, que será emitido quando um usuário entrar no chat. Implementaremos um simples condicional para criar o *hash* da sala quando não existir e em seguida armazenamos em memória via `client.set('sala', sala)`.

```
client.on('join', function(sala) {
  if(sala) {
    sala = sala.replace('?','');
  } else {
    var timestamp = new Date().toString();
    var md5 = crypto.createHash('md5');
    sala = md5.update(timestamp).digest('hex');
  }
  client.set('sala', sala);
  client.join(sala);
});
```

Para controlar a saída de usuários na sala, vamos utilizar o evento *default* do Socket.IO chamado de `client.on('disconnect')`, que será utilizado para excluir

um usuário da sala. Para descobrirmos em qual sala o usuário esta conectado usaremos a função `client.get('sala')` para recuperar a sala, e em seu callback iremos remover o usuário através da função: `client.leave(sala)`.

```
client.on('disconnect', function () {
  client.get('sala', function(erro, sala) {
    client.leave(sala);
  });
});
```

Para finalizar, vamos atualizar o evento `client.on('send-server')`, para que ele envie uma mensagem somente para usuários de uma sala através da função `sockets.in(sala).emit('send-client', msg)`. Também vamos criar um novo evento chamado: `client.broadcast.emit('new-message', data);` — sua variável `data` terá como parâmetro o e-mail e sala do cliente e ele será executado para atualizar a url do botão **'Conversar'** do contato que receber uma mensagem.

```
client.on('send-server', function (msg) {
  var msg = "<b>"+ usuario.nome +":</b> "+ msg +"<br>";
  client.get('sala', function(erro, sala) {
    var data = {email: usuario.email, sala: sala};
    client.broadcast.emit('new-message', data);
    sockets.in(sala).emit('send-client', msg);
  });
});
```

Com o nosso back-end implementado, resta-nos preparar o front-end. Precisamos incluir o botão **'Conversar'** enviando em sua url o *hash* da sala. Com isso, vamos criar um novo ponto de conexão com Socket.IO. Este ponto será implementado na agenda de contatos em `views/contatos/index.ejs`. Crie um novo *partial* chamado `notify_script.ejs`, com o seguinte código:

```
<script src="/socket.io/socket.io.js"></script>
<script>
  var socket = io.connect('http://localhost:3000');
  socket.on('new-message', function(data) {
    var chat = document.getElementById('chat_' + data.email);
    chat.href += '?' + data.sala;
  });
</script>
```

Agora, atualize a view `contatos/index.ejs` para incluir este *partial*. Também iremos inserir o botão **'Conversar'**, permitindo que o nosso chat aconteça através da nossa agenda de contatos:

```
<% include ../header %>
<header>
  <h2>Ntalk - Agenda de contatos</h2>
</header>
<section>
  <!-- Formulário de novo contato -->
  <table>
    <thead>
      <tr>
        <th>Nome</th>
        <th>E-mail</th>
        <th>Ação</th>
      </tr>
    </thead>
    <tbody>
      <% contatos.forEach(function(contato, index) { %>
        <tr>
          <td><%- contato.nome %></td>
          <td><%- contato.email %></td>
          <td>
            <a href="/contato/<%- index %>">Detalhes</a>
            <a href="/chat/<%- contato.email %>"
               id="chat_<%- contato.email %>">
              Conversar
            </a>
          </td>
        </tr>
      <% }) %>
    </tbody>
  </table>
</section>
<% include notify_script %>
<% include ../exit %>
<% include ../footer %>
```

Reinicie o servidor e faça o teste! Se tudo der certo, significa que implementamos um chat integrado com os usuários de nossa agenda de contatos.

6.7 Notificadores na agenda de contatos

Para finalizar este capítulo com chave de ouro, vamos criar um simples notificador na agenda de contatos. Esse notificador vai informar o status de cada contato, que terá apenas três estados: **Online**, **Offline** e **Mensagem**. Visualmente ele será um novo campo na tabela de contatos e iremos explorar novas funções do Socket.IO para torná-lo real-time.

Abra o código `sockets/chat.js` e implemente no início do evento `sockets.on('connection')`, um nova regra que seguirá a seguinte lógica: armazenar o e-mail do usuário online e depois rodar um loop que contém todos os usuários conectados para que em cada iteração seja enviado os e-mails via função: `client.emit('notify-onlines', email)` para notificar próprio usuário e também através da função: `client.broadcast.emit('notify-onlines', email)`, atualizando os demais usuários conectados. Veja como faremos isso no código a seguir:

```
sockets.on('connection', function (client) {
  var session = client.handshake.session
    , usuario = session.usuario;
  client.set('email', usuario.email);

  var onlines = sockets.clients();
  onlines.forEach(function(online) {
    var online = sockets.sockets[online.id];
    online.get('email', function(err, email) {
      client.emit('notify-onlines', email);
      client.broadcast.emit('notify-onlines', email);
    });
  });

  // continuação dos eventos...
});
```

No Socket.IO temos a função que permite que esta brincadeira aconteça. Ela se chama: `sockets.clients()` e retorna um *array* contendo os ids dos clientes conectados. Com base no `id`, retornamos um cliente através da função: `sockets.sockets[online.id]` e em seguida pegamos o seu email (`online.get('email')`). Neste projeto temos dois pontos de conexões com Socket.IO: **agenda de contatos e chat**. Sendo assim, fica inviável utilizar esta lógica

utilizando o `client.id`, pois este id é autogerado a cada vez que entramos e saímos de um ponto de conexão. Por causa desse motivo utilizamos o e-mail do usuário como identificador chave através da função `client.set('email', usuario.email)` antes de rodar o loop, possibilitando iterar o *array* de clientes cujo atributo `store`, que contém os dados do usuário.

Já temos um notificador para usuários onlines, agora vamos criar os notificadores para os status: **Offline** e **Mensagem**. Não há segredo para implementá-los, apenas temos que reutilizar a função `client.broadcast.emit('new-message')` para atualizar o status de **Mensagem** no usuário e implementar a função `client.broadcast.emit('notify-offline')` dentro do evento `client.on('disconnect')`, seguindo o código abaixo:

```
client.on('send-server', function (msg) {
  var msg = "<b>"+ usuario.nome +":</b> "+ msg +"<br>";
  client.get('sala', function(erro, sala) {
    var data = {email: usuario.email, sala: sala};
    client.broadcast.emit('new-message', data);
    sockets.in(sala).emit('send-client', msg);
  });
});

client.on('disconnect', function() {
  client.get('sala', function(erro, sala) {
    var msg = "<b>"+ usuario.nome +":</b> saiu.<br>";
    client.broadcast.emit('notify-offline', usuario.email);
    sockets.in(sala).emit('send-client', msg);
    client.leave(sala);
  });
});
```

Com o back-end desenvolvido, vamos finalizar esta tarefa codificando o comportamento da view `contatos/index.ejs` para que sejam renderizados os status na lista de contatos, vamos fazer essas modificações na view `contatos/notify_script.ejs`:

```
<script src="/socket.io/socket.io.js"></script>
<script>
  var socket = io.connect('http://localhost:3000');
  var notify = function(data) {
    var id = 'notify_' + data.el;
```

```
    var notify = document.getElementById(id);
    notify.textContent = data.msg;
  };
  socket.on('notify-onlines', function(email) {
    notify({el: email, msg: 'Online'});
  });
  socket.on('notify-offline', function(email) {
    notify({el: email, msg: 'Offline'});
  });
  socket.on('new-message', function(data) {
    notify({el: data.email, msg: 'Mensagem'});
    var id = 'chat_' + data.email;
    var chat = document.getElementById(id);
    chat.href += '?' + data.sala;
  });
</script>
```

Com isso implementado, basta atualizar a lista de contatos para visualizar as mensagens de notificação, edite a view `contatos/index.ejs` incluindo uma tag `<span id="notify_<%- contato.email %>">` para que o código javascript anterior faça a magia!

```
<% contatos.forEach(function(contato, index) { %>
  <tr>
    <td><%- contato.nome %></td>
    <td><%- contato.email %></td>
    <td><span id="notify_<%- contato.email %>">Offline</span></td>
    <td>
      <a href="/contato/<%- index %>">Detalhes</a>
      <a href="/chat/<%- contato.email %>"
         id="chat_<%- contato.email %>">
        Conversar
      </a>
    </td>
  </tr>
<% }) %>
```

Agora temos o nosso notificador pronto! Para testá-lo, reinicie o servidor, crie 3 contas no Ntalk cadastrando os e-mails de cada conta como contato entre elas para possibilitar uma conversa no chat. Por exemplo, cadastro da conta A, B, C e os contatos da conta A são os usuários da conta B e C e assim faça o mesmo com as demais

para cria uma rede de contatos.

Depois disso converse no chat a conta A com a B, repare que agora os status vão se alterar em tempo real de acordo com a interação do usuário.

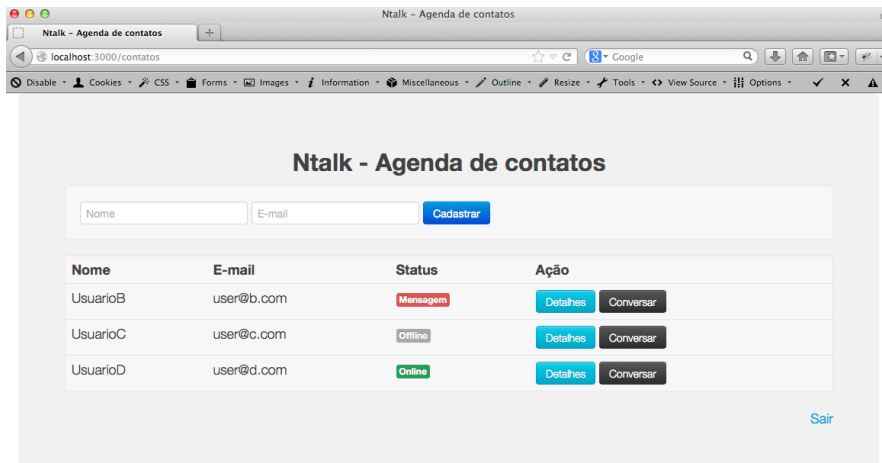



Figura 6.7: Notificações da agenda de contatos do Usuário A.

6.8 Principais eventos do Socket.IO

Para complementar seus estudos com este módulo, apresentarei abaixo os principais eventos do *Socket.IO*, tanto no servidor como para cliente.

No lado do servidor:

- `io.sockets.on('connection', function(client))` - Evento que acontece quando um novo cliente se conecta no servidor.
- `client.on('message', function(mensagem, callback))` - Ocorre quando um cliente se comunica através da função `send()`, o *callback* desse evento responde automaticamente o cliente no final de sua execução.
- `client.on('qualquer-nome-de-evento', function(data))` - São eventos criados pelo desenvolvedor, qualquer nome pode ser apelidado aqui, exceto os nomes dos eventos principais e o seu comportamento é de apenas receber objetos através da variável `data`. Em nosso chat criamos o evento `'send-server'`.

- `client.on('disconnect', callback)` - Quando um cliente sai do sistema é emitido o evento `'disconnect'` para o servidor. Também é possível emitir esse evento no cliente sem precisar sair do sistema.

No lado do cliente:

- `client.on('connect', callback)` – Ocorre quando o cliente se conecta no servidor.
- `client.on('connecting', callback)` – Ocorre quando o cliente está se conectando no servidor.
- `client.on('disconnect', callback)` – Ocorre quando o cliente se desconecta do servidor.
- `client.on('connect_failed', callback)` – Ocorre quando o cliente não conseguiu se conectar no servidor devido a falhas de comunicação entre cliente com servidor.
- `client.on('error', callback)` – Ocorre quando o cliente já se conectou, porém um erro no servidor ocorreu durante as trocas de mensagens.
- `client.on('message', function(message, callback))` – Ocorre quando o cliente envia uma mensagem de resposta rápida ao servidor, cujo o retorno acontece através da função de *callback*.
- `client.on('qualquer-nome-de-evento', function(data))` – Evento customizado pelo desenvolvedor. No exemplo do web chat criamos o evento `'send-client'` que envia mensagem para o servidor.
- `client.on('reconnect_failed', callback)` – Ocorre quando o cliente não consegue se reconectar no servidor.
- `client.on('reconnect', callback)` – Ocorre quando o cliente se reconecta ao servidor.
- `client.on('reconnecting', callback)` – Ocorre quando o cliente está se reconectando no servidor.

E mais uma vez implementamos uma incrível funcionalidade em nosso sistema. No próximo capítulo iremos otimizar a agenda de contatos adicionando um banco de dados para persistir os contatos dos usuários e também incluiremos um histórico de conversas no chat.

Capítulo 7

Integração com banco de dados

7.1 Bancos de dados mais adaptados para Node.js

Nos capítulos anteriores (em especial o capítulo 4, 5 e 6) aplicamos um modelo simples de banco de dados, mais conhecido como *MemoryStore*. Ele não é um modelo adequado de persistência de dados, pois quando o usuário sair da aplicação ou o servidor for reiniciado, todos os dados serão apagados. Utilizamos esse modelo apenas para apresentar os conceitos sobre os módulos Express e Socket.IO.

Neste capítulo, vamos aprofundar nossos conhecimentos trabalhando com um banco de dados de verdade para Node.js. Algo fortemente ligado ao Node.js são os banco de dados NoSQL. É claro que existem módulos de banco de dados SQL, mas de fato, módulos NoSQL são mais populares nesta plataforma.

A grande vantagem de trabalhar com esse modelo de banco de dados é a grande compatibilidade e suporte mantido pela comunidade própria Node.js. Os NoSQL populares são: MongoDB (http://www.mongodb.org) , Redis (http://redis.io/) , CouchDB (http://couchdb.apache.org) e RiakJS (http://riakjs.com) . Dos bancos de da-

dos SQL existem alguns módulos para MySQL (http://www.mysql.com) , SQLite (http://www.sqlite.org) e Postgre (http://www.postgresql.org) .

Caso queira ver todos os *drivers* compatíveis com Node.js veja este link:

https://github.com/joyent/node/wiki/Modules#wiki-database

Figura 7.1: NoSQL MongoDB.

Neste livro, utilizaremos o MongoDB, ele é um banco de dados NoSQL, mantido pela empresa 10gen e foi escrito em linguagem C/C++. Ele utiliza Javascript como interface para manipulação de dados e a persistência dos dados é feita através de objetos JSON. Nele trabalhamos com o conceito *schema-less*, ou seja, não existe relacionamentos de tabelas, nem chaves primárias ou estrangeiras e sim *documents* que possuem *embedded documents* e tudo mantido dentro de uma *collection*. Outra vantagem do *schema-less* é que os atributos são inseridos ou removidos em *runtime*, sem a necessidade de travar uma *collection*, tornando este banco de dados flexível a grandes mudanças. Como disse antes, com o MongoDB podemos persistir *embedded documents* dentro de um *document*, que seria o mesmo que criar relacionamento entre tabelas, porém neste conceito tudo é inserido em uma mesma tabela (em um mesmo *document*). Isso diminui o número de consultas complexas no banco de dados e principalmente evita criar *joins* para carregar diversas informações de uma vez.

Não vamos entrar em detalhes sobre como instalar o MongoDB ou utilizá-lo. Para simplificar o nosso aprendizado utilizaremos as configurações padrões que já vem ao instalá-lo. Para instalar e também conhecer mais a fundo esse banco de dados visite seu site oficial:

http://mongodb.com

7.2 MongoDB no Node.js utilizando Mongoose

O Mongoose possui uma interface muito fácil de aprender. Em poucos códigos, você vai conseguir criar uma conexão no banco de dados e executar uma query ou persistir dados. Com o MongoDB instalado e funcionando em sua máquina, vamos instalar o módulo `mongoose`, que é um framework responsável por mapear objetos do Node.js para MongoDB. Atualize no `package.json`:

```
"dependencies": {
  "express": "3.4.7",
  "express-load": "1.1.8",
  "ejs": "0.8.5",
  "socket.io": "0.9.16",
  "mongoose": "3.8.4"
}
```

E em seguida execute o comando:

```
npm install
```

Para a aplicação se conectar com o banco de dados, no `app.js`, utilizaremos a variável `db` em modo global para manter uma conexão com o banco de dados compartilhando seus recursos em todo projeto:

```
var express = require('express')
  , app = express()
  , load = require('express-load')
  , error = require('./middleware/error')
  , server = require('http').createServer(app)
  , io = require('socket.io').listen(server)
  , mongoose = require('mongoose');

global.db = mongoose.connect('mongodb://localhost/ntalk');
```

Quando é executada a função `mongoose.connect` cria-se uma conexão com o banco de dados MongoDB para o Node.js. Como o MongoDB é *schemaless*, na primeira vez que a aplicação se conecta com o banco através da url `'mongodb://localhost/ntalk'` automaticamente em *run-time* é criada uma base de dados com o nome **ntalk**.

7.3 Modelando com Mongoose

O Mongoose é um módulo focado para a criação de *models*, isso significa que com ele criaremos objetos persistentes modelando seus atributos através do objeto `mongoose.Schema`. Após a implementação de um modelo, temos que registrá-lo no banco de dados, utilizando a função `db.model('nome-do-model', modelSchema)`, que recebe um modelo e cria sua respectiva `collection` no MongoDB.

Vamos explorar as principais funcionalidades do Mongoose aplicando na prática em nosso projeto. Com isso, faremos diversos *refactorings* em toda aplicação para substituir o modelo de persistência via *session* para o modelo do Mongoose que armazena dados no MongoDB.

Para começar vamos criar o modelo `usuario.js` no diretório *models*. Ele será o modelo principal e terá os seguintes atributos: `nome`, `email` e `contatos`. A modelagem dos atributos acontece através do objeto `require('mongoose').Schema`, veja abaixo como ficará esta modelagem:

```
module.exports = function(app) {
  var Schema = require('mongoose').Schema;

  var contato = Schema({
     nome: String
   , email: String
  });
  var usuario = Schema({
     nome: {type: String, required: true}
   , email: {type: String, required: true
                , index: {unique: true}}
   , contatos: [contato]
  });

  return db.model('usuarios', usuario);
};
```

Repare que foram criados dois objetos `usuario` e `contato`, e apenas foi registrado o modelo `usuario`, pois `contato` será um subdocumento de `usuario` e o registro ocorre via função `app.db.model()`. Outro detalhe importante: incluímos dois tipos de validações neste modelo, que são: `required` e `unique`. Para quem não conhece o `required: true` valida se o seu atributo possui algum valor, ou

seja, ele não permite persistir um campo vazio. Já o `unique: true` cria um índice de valor único em seu atributo, semelhante nos bancos de dados SQL. Estas validações geram um erro que é enviado no callback de qualquer função de persistência do modelo, por exemplo, `usuario.create()` ou `usuario.update()`.

7.4 Implementando um CRUD na agenda de contatos

Com o modelo implementado, o que nos resta a fazer é alterar os controllers para utilizarem suas funções. Começando do mais fácil, vamos modificar o controller `home.js`. Nele, executaremos a função `findOne`, que retorna apenas um objeto, e a função `select('name email')`, que filtra esse objeto retornando um novo objeto contendo apenas os atributos `name` e `email`. Com isso, evita-se que seja carregado o subdocumento `contatos`. Na prática, esta query seria algo que em banco de dados SQL faríamos com o seguinte comando: `SELECT * FROM usuario LIMIT 1`. Caso não seja encontrado um usuário, cadastraremos um novo através da função `Usuario.create`. Veja abaixo o código-fonte:

```
login: function(req, res) {
  var query = {email: req.body.usuario.email};
  Usuario.findOne(query)
         .select('nome email')
         .exec(function(erro, usuario){
    if (usuario) {
      req.session.usuario = usuario;
      res.redirect('/contatos');
    } else {
      Usuario.create(req.body.usuario, function(erro, usuario) {
        if(erro){
          res.redirect('/');
        } else {
          req.session.usuario = usuario;
          res.redirect('/contatos');
        }
      });
    }
  });
}
```

Repare que com o *Mongoose* é possível criar queries complexas chamando funções que em seu retorno permite chamar uma outra função. Isso forma um encade-

amento de funções e o final desse encadeamento ocorre quando chamamos a função `exec()`. Um bom exemplo disso é a query abaixo:

```
Usuario.findOne(req.body.usuario)
       .select('nome email')
       .exec(function(erro, usuario) {
  // continuação do código...
});
```

A função `Usuario.create` persiste um objeto que tenha os mesmo atributos de seu modelo, caso contrário ocorrerá um erro e os detalhes dele será enviado para variável `erro` presente no callback. Por este motivo é recomendável sempre tratar esses erros para garantir estabilidade no sistema.

Parabéns! Com o controle de login funcionando corretamente, só falta modificar o controller `contatos.js` e suas respectivas views. Nesta etapa exploraremos novas funções do MongoDB através do modelo `Usuario`, veja abaixo as mudanças de cada *action*:

```
index: function(req, res) {
  var _id = req.session.usuario._id;
  Usuario.findById(_id, function(erro, usuario) {
    var contatos = usuario.contatos;
    var resultado = { contatos: contatos };
    res.render('contatos/index', resultado);
  });
}
```

Na *action* `index` utilizamos a função `Usuario.findById()` que retorna apenas um usuário baseado no `_id` em parâmetro, essa função será o suficiente para retornar os dados do usuário e todos os seus contatos.

```
create: function(req, res) {
  var _id = req.session.usuario._id;
  Usuario.findById(_id, function(erro, usuario) {
    var contato = req.body.contato;
    var contatos = usuario.contatos;
    contatos.push(contato);
    usuario.save(function() {
      res.redirect('/contatos');
    });
  });
}
```

Já na *action* `create`, temos apenas que atualizar a lista de contatos incluindo um novo contato. Para isso, buscamos o usuário via função `Usuario.findById()` e em seu callback atualizamos o *array* `usuario.contatos` com a função: `contatos.push(contato)` e em seguida é executado a função `usuario.save()`.

```
show: function(req, res) {
  var _id = req.session.usuario._id;
  Usuario.findById(_id, function(erro, usuario) {
    var contatoID = req.params.id;
    var contato = usuario.contatos.id(contatoID);
    var resultado = { contato: contato };
    res.render('contatos/show', resultado);
  });
},
edit: function(req, res) {
  var _id = req.session.usuario._id;
  Usuario.findById(_id, function(erro, usuario) {
    var contatoID = req.params.id;
    var contato = usuario.contatos.id(contatoID);
    var resultado = { contato: contato };
    res.render('contatos/edit', resultado);
  });
}
```

As *actions* `show` e `edit` possuem o mesmo comportamento. Eles retornam os dados de um específico contato do usuário através da função `usuario.contatos.id(contatoID)` que é uma função do *embedded document* `contatos` que retorna um contato baseado em seu `_id`.

```
update: function(req, res) {
  var _id = req.session.usuario._id;
  Usuario.findById(_id, function(erro, usuario) {
    var contatoID = req.params.id;
    var contato = usuario.contatos.id(contatoID);
    contato.nome = req.body.contato.nome;
    contato.email = req.body.contato.email;
    usuario.save(function() {
      res.redirect('/contatos');
    });
```

```
  });
}
```

Nesta *action*, a implementação do update não tem segredos. Como contatos é um *embedded document*, seu tratamento é semelhante às funções de um array, por isso praticamente buscamos o usuário de acordo com seu _id via função Usuario.findById(). Em seguida, buscamos o seu respectivo contato com base em seu contatoID, e atualizamos seus atributos normalmente. Quando executamos a função usuario.save(), o contato será atualizado na base de dados.

```
destroy: function(req, res) {
  var _id = req.session.usuario._id;
  Usuario.findById(_id, function(erro, usuario) {
    var contatoID = req.params.id;
    usuario.contatos.id(contatoID).remove();
    usuario.save(function() {
      res.redirect('/contatos');
    });
  });
}
```

Em destroy, temos três ações para excluir um contato. Primeiro, buscamos um contato via função Usuario.findById(), em seguida, buscamos um específico contato do usuário já o excluindo através da linha usuario.contatos.id(contatoID).remove() e para finalizar atualizamos os dados do usuário executando: usuario.save().

Para finalizar vamos atualizar as views do contato. A edição será bem simples, vamos apenas trocar a maneira como eles renderizam o id do contato nas urls. Veja abaixo como faremos esta atualização. Abra as views contatos/edit.ejs e contatos/show.ejs, neles mude apenas a url da action de seus respectivos form, renderizando o atributo contato._id:

```
<form action="/contato/<%- contato._id %>" method="post">
```

Agora na view contatos/index.ejs alteraremos as urls dos link gerados na iteração do loop de contatos:

```
<% contatos.forEach(function(contato) { %>
  <tr>
    <td><%- contato.nome %></td>
```

```
    <td><%- contato.email %></td>
    <td>
      <span id="notify_<%- contato.email %>" class="label">
        Offline
      </span>
    </td>
    <td>
      <a href="/contato/<%- contato._id %>">Detalhes</a>
      <a href="/chat/<%- contato.email %>"
         id="chat_<%- contato.email %>">Conversar
      </a>
    </td>
  </tr>
<% }) %>
```

Com as views finalizadas terminamos o nosso *refactoring* na agenda de contatos. Para verificar se tudo ocorreu bem, reinicie o servidor e confira as novidades no sistema. Dessa vez temos um sistema integrado ao MongoDB, persistindo seus contatos, cadastrando usuário e fazendo login corretamente.

7.5 Persistindo estruturas de dados com NoSQL Redis

Com nossa agenda integrada ao MongoDB e persistindo contatos no banco de dados, precisamos agora persistir dados das conversas do chat da aplicação, para que ele mantenha um histórico de conversas.

Como o chat é será uma área de maior acesso, precisamos armazenar seus dados em uma estrutura simples de chave-valor e que permita leituras muito rápido. Para manter este tipo de estrutura o MongoDB não seria uma boa solução, pois precisamos de um banco de dados que mantenha dados frequentemente em memória para garantir uma leitura rápida deles. Ele se chama Redis.

Figura 7.2: NoSQL Redis.

O Redis guarda e busca, em sua base de dados, elementos chave-valor, de maneira extremamente rápida, pois mantém os dados em grande parte do tempo na memória, fazendo em curtos períodos a sincronização dos dados com disco rígido. Ele é considerado um NoSQL do tipo em chave-valor, em que a chave é o identificador e o valor pode ser diversos tipos de estruturas de dados.

As estruturas de dados que ele trabalha são: *Strings*, *Hashes*, *Lists*, *Sets* e *Sorted Sets*. E ele possui um CLI (através do comando `redis-cli`) que permite em *runtime* brincar com seus inúmeros comandos — aliás, são vários comandos que realizam operações com os dados, vale a pena dar uma olhada em sua documentação:

http://redis.io/commands

7.6 Mantendo um histórico de conversas do chat

Utilizaremos o Redis para implementar o histórico de conversas do nosso chat. Basicamente vamos persistir cada mensagem em uma lista, agrupando-a em uma chave que será o `id` da sala do chat. Isso vai fazer com que o usuário que receber uma mensagem consiga visualizá-la após clicar no botão **"Conversar"**.

Assim como foi no MongoDB, não entraremos em detalhes sobre como instalar e configurar o Redis. E todos os exemplos deste livro foi utilizado a configuração padrão dele. Para baixar e instalar o Redis visite este link:

http://redis.io/download

Já considerando que o Redis esta instalando e funcionando corretamente em sua máquina, instalaremos seu *driver* compatível com Node.js e um módulo adicional, é um *wrapper* que aumenta performance do Redis com Node.js, permitindo processamento assíncrono e outras otimizações de baixo nível nas operações do Redis, ele

se chama *Hi-Redis*, apenas instale-o que o próprio módulo Redis utilizará automaticamente. No terminal, execute o comando:

```
npm install redis hiredis --save
```

Com o Redis e seu respectivo driver instalados corretamente, vamos ao que interessa, que é implementar o histórico do chat. Abra o código `sockets/chat.js`, será nele que iremos conectar o driver com o servidor Redis para habilitar seus comandos na aplicação. É a partir da execução de `redis = require('redis').createClient()`, que começa tudo. Praticamente, ele carrega o driver e retorna um cliente Redis. Como o banco Redis e a aplicação Node.js estão hospedados na mesma máquina e utilizamos as configurações padrões do Redis, então não há necessidade de enviar parâmetros extras para função `createClient()`. Caso você necessite conectar de um local diferente, apenas inclua os seguintes parâmetros: `createClient(porta, ip)`. Veja abaixo como será o nosso código:

```
module.exports = function(io) {

  var crypto = require('crypto')
    , redis = require('redis').createClient()
    , sockets = io.sockets;

  // continuação dos eventos do socket.io...
}
```

Agora com um cliente Redis em ação, vamos implementar algumas de suas funções para pesquisar e persistir as mensagens. Utilizaremos a estrutura lista para armazenar as mensagens. Cada lista terá uma sala como chave para pesquisa. Primeiro vamos implementá-la no evento `client.on('join')`:

```
client.on('join', function(sala) {
  if(sala){
    sala = sala.replace('?','');
  } else {
    var timestamp = new Date().toString();
    var md5 = crypto.createHash('md5');
    sala = md5.update(timestamp).digest('hex');
  }
  client.set('sala', sala);
```

```
  client.join(sala);

  var msg = "<b>"+usuario.nome+":</b> entrou.<br>";
  redis.lpush(sala, msg, function(erro, res) {
    redis.lrange(sala, 0, -1, function(erro, msgs) {
      msgs.forEach(function(msg) {
        sockets.in(sala).emit('send-client', msg);
      });
    });
  });
});
```

Basicamente utilizamos duas de suas funções: primeiro executamos a função `redis.lpush(sala, msg)` que adiciona na lista a mensagem. Depois em seu callback utilizamos a função `redis.lrange(sala, 0, -1)` que retorna um array contendo os elementos a partir de um range inicial e final da lista. O range utiliza dois índices, e neste caso o índice inicial é `0` e o final é `-1`. Quando informamos o valor `-1` no índice final indicamos que o range será total, retornando todos os elementos da lista. Por último, no callback do `redis.lrange()` iteramos o array de mensagens emitindo mensagem por mensagem para o cliente.

Agora para finalizar o nosso histórico do chat, implementaremos a função `redis.lpush` nos eventos `send-server` e `disconnect`. Como estes eventos enviam uma única mensagem, simplificaremos o código incluindo a função `redis.lpush(sala, msg)` sem utilizar callbacks:

```
client.on('send-server', function (msg) {
  var msg = "<b>"+ usuario.nome +":</b> "+ msg +"<br>";
  client.get('sala', function(erro, sala) {
    redis.lpush(sala, msg);
    var data = {email: usuario.email, sala: sala};
    client.broadcast.emit('new-message', data);
    sockets.in(sala).emit('send-client', msg);
  });
});

client.on('disconnect', function() {
  client.get('sala', function(erro, sala) {
    var msg = "<b>"+ usuario.nome +":</b> saiu.<br>";
    redis.lpush(sala, msg);
    client.broadcast.emit('notify-offline', usuario.email);
```

```
    sockets.in(sala).emit('send-client', msg);
    client.leave(sala);
  });
});
```

E mais uma vez terminamos um excelente capítulo! Agora temos uma aplicação 100% funcional que utiliza dois banco de dados NoSQL. Acredite, o que já temos aqui já é o suficiente para colocar a aplicação no ar. Mas continue lendo, afinal nos próximos capítulos vamos aprofundar nossos conhecimentos codificando testes e também otimizando o sistema para que ele entre em um ambiente de produção de forma eficiente.

Capítulo 8

Preparando um ambiente de testes

8.1 Mocha, o framework de testes para Node.js

Testes automatizados é algo cada vez mais adotado no mundo de desenvolvimento de sistemas. Existem diversos tipos de testes: teste unitário, teste funcional, teste de aceitação, entre outros. Neste capítulo focaremos apenas no teste de aceitação, para o qual temos alguns frameworks para realizá-lo. O mais recente e que anda ganhando visibilidade pela comunidade, é o Mocha, seu site é

http://visionmedia.github.io/mocha

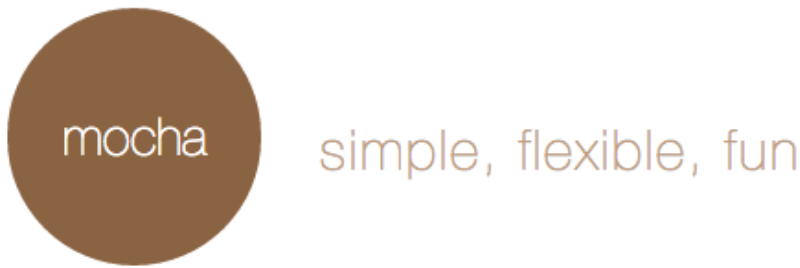



Figura 8.1: Mocha - Framework para testes

Ele é mantido pelo mesmo criador do Express (TJ Holowaychuk), e foi criado com as seguintes características: teste no estilo TDD, testes no estilo BDD, cobertura de código, relatório em HTML, teste de comportamento assíncrono, integração com os módulos: `should` e `assert`. Praticamente, ele é um ambiente completo para desenvolvimento de testes. Possui diversas interfaces de apresentação do resultado dos testes. Nas seções a seguir, apresentarei o Mocha, desde a sua configuração até a implementação de testes no projeto Ntalk.

8.2 Criando um Environment para testes

Antes de entrarmos a fundo nos testes, primeiro temos que criar um novo ambiente com configurações específicas para testes. Isso envolve criar uma função que contenha informações para se conectar em uma banco de dados de testes e de desenvolvimento. Com isso vamos migrar a função `mongoose.connect` para este novo arquivo, para que ele retorne uma conexão de banco de dados de acordo com o ambiente. Para identificar em qual ambiente esta o projeto, utilizamos a variável `process.env.NODE_ENV`.

Vamos criar um novo diretório, responsável por manter bibliotecas externas, ela se chamará de `lib`, nele vamos criar o arquivo `db_connect.js` e inserir a lógica abaixo:

```
module.exports = function() {
  var mongoose = require('mongoose');
  var env_url = {
      "test": "mongodb://localhost/ntalk_test"
```

```
    , "development": "mongodb://localhost/ntalk"
  };
  var url = env_url[process.env.NODE_ENV || "development"];
  return mongoose.connect(url);
};
```

Lendo variáveis de ambiente no Node

Quando se trabalha com variáveis de ambiente é muito comum persistir dados de configurações no sistema operacional. Url de acesso a banco de dados ou outros serviços, assim como senhas e chaves importantes de acesso a sistemas externos são alguns exemplos de variáveis de ambiente. Esses dados são configurados em um arquivo no próprio sistema operacional. No capítulo 1 foi explicado como criar a variável `NODE_ENV`, para outras variáveis se faz o mesmo procedimento. E no Node.js podemos lê-las através do `process.env["VARIAVEL"]`, que é um objeto JSON que contém todas as variáveis do sistema operacional.

Com essa função preparada, podemos remover o carregamento do `mongoose` e sua variável `global.db` dentro do `app.js`. Afinal quanto menos variáveis globais existirem na aplicação melhor. Como preparamos uma *lib* que retorna uma conexão de acordo com a variável de ambiente `NODE_ENV`, o carregamento do `db_connect.js` será diretamente no modelo `models/usuario.js`:

```
module.exports = function(app) {
  var db = require('../lib/db_connect')()
    , Schema = require('mongoose').Schema;

  var contato = Schema({
      nome: String
    , email: String
  });
  var usuario = Schema({
      nome: { type: String, required: true }
    , email: { type: String, required: true
             , index: {unique: true} }
    , contatos: [contato]
```

```
  });

  return db.model('usuarios', usuario);
};
```

Dessa forma mantemos nossa aplicação com nenhuma variável global, e a variável `db` que mantém uma conexão com MongoDB será gerenciada pelo `db_connect.js`.

Pronto! Essa foi uma demonstração simples de como sua aplicação vai se autoconfigurar. Com essas configurações ela estará preparada para desenvolvimento multi-ambiente. A princípio só criamos uma *lib* que retorna uma instância de conexão com banco de dados de acordo com sua variável de ambiente, mas em aplicações mais complexas, utiliza-se muito desse conceito para criar outros tipos de *libs*.

8.3 Instalando e configurando o Mocha

Para começarmos com o Mocha vamos instalá-lo em modo global para a utilização do seu CLI. Em seu console execute o comando:

```
npm install -g mocha
```

O Mocha é um módulo focado em testes e vamos adicioná-lo no `package.json`, porém ele não será incluído dentro de `dependencies`. O motivo é que de fato ele é muito pesado e não é um framework para ser carregado em um ambiente de produção. Neste caso, existe um outro atributo chamado `devDependencies`, que é utilizado para integrar módulos específicos para automatização de testes e deploy.

```
"devDependencies": {
  "mocha": "*"
}
```

Na seção seguinte implementaremos alguns testes utilizando interface BDD *(Behavior Driven-Development)* para usar funções como `describe`, `it`, `beforeEach`, `should` e outras. A função `should` faz verificações em cima dos resultados de cada testes, porém ela não é nativa do framework Mocha, com isso teremos que habilitá-la instalando o seu próprio módulo, o `should`.

```
"devDependencies": {
  "mocha": "*",
```

```
  "should": "*"
}
```

Para explorarmos o Mocha, implementaremos apenas testes funcional sobre as rotas da aplicação. Para realizar esse tipo de teste precisamos de um módulo que faça requisições em nosso servidor. Testar requisições sobre as rotas é muito útil, pois permite verificar como será o comportamento de uma requisição feita por um usuário. Para realizar esses testes, utilizaremos o módulo `supertest` que também nasceu pelo os mesmos criadores do Mocha. Adicione esse módulo no `package.json`, seguindo o código abaixo:

```
"devDependencies": {
  "mocha": "*",
  "should": "*",
  "supertest": "*"
}
```

Para finalizar, crie o diretório `test`, que será o local em que codificaremos os testes da aplicação, e instale todos os módulos através do comando: `npm install`.

8.4 Rodando o Mocha no ambiente de testes

Assim como criamos o `db_connect.js`, que é um simples script que retorna uma conexão MongoDB baseado no valor da variável `NODE_ENV`, temos que executar os testes utilizando esta variável com valor `'test'` para rodar os testes no seu devido ambiente. Para executar testes com o Mocha, um simples comando no terminal: `mocha test` já será o suficiente, mas neste caso ele não será executado no ambiente de testes. O comando correto é `NODE_ENV=test mocha test`, pois ele define o valor de `NODE_ENV` para `test`. Mas executar esse comando longo seria um pouco cansativo, e como um bom programador tem que ser preguiçoso, que tal simplificá-lo?

Uma boa prática para simplificar este comando é utilizar o `package.json` para definir comandos dentro do atributo `"scripts"`. Ele será convertido para um comando executável via `npm` — por *default*, já existe um comando no `npm` para executar este tipo de tarefa. Veja o código abaixo, que será incluído dentro do `package.json`:

```
"scripts": {
  "start": "node app",
```

```
  "test": "NODE_ENV=test ./node_modules/mocha/bin/mocha test/*.js"
}
```

Porque usamos o mocha da pasta node_modules?

Dentro do nosso projeto, cada dependência dele fica localizada dentro da pasta `node_modules`. Apenas **módulos globais**, ficam fora desta pasta, afinal eles são armazenados em uma pasta específica que varia de acordo com o sistema operacional. Quando utilizamos os comandos `npm test` ou `npm start`, qualquer utilização de módulo deve obrigatoriamente ser chamada dentro do diretório `node_modules`, porque este comando geralmente é utilizado por serviços de terceiros. Um bom exemplo de serviço é o Travis CI (http://travis-ci.org) , um serviço de *deploy contínuo* compatível com diversas linguagens, inclusive Node.js. Ele roda os testes do seu projeto através do comando `npm test` do seu `package.json`. Ele não instala módulos globais, apenas utiliza os módulos existentes no `node_modules`.

Foram adicionados dois `"scripts"`, o `"start"` e o `"test"`. Estes são os comandos atalhos do `npm`, ou seja, agora os testes serão executados via comando `npm test` e a aplicação será iniciada via comando `npm start`. Dentro de `scripts` você pode criar quantos comandos quiser e todos eles serão executados via comando: `npm run [nome-do-comando]`.

8.5 Testando as rotas

O módulo `supertest` será intensivamente usado e antes de criarmos os testes temos que exportar a variável `app` do `app.js` para que automaticamente levante uma instância de servidor para os testes. Este `refactoring` é muito simples, apenas inclua na última linha do `app.js` o seguinte trecho:

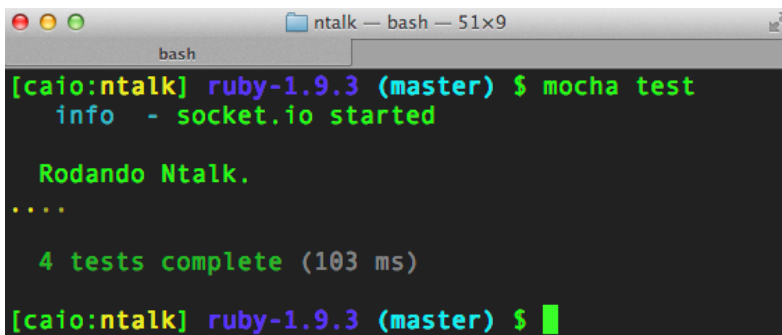
```
// Trecho final do app.js...
module.exports = app;
```

Feito isso agora podemos criar os testes. Dentro do diretório `test` crie o arquivo `home.js`. No primeiro teste simularemos uma requisição para a rota principal "/" esperando que retorne o status 200 como sucesso da requisição.

```
var app = require('../app')
  , should = require('should')
  , request = require('supertest')(app);

describe('No controller home', function() {
  it('deve retornar status 200 ao fazer GET /', function(done){
    request.get('/')
           .end(function(err, res){
      res.status.should.eql(200);
      done();
    });
  });
  // continuação...
```

No mesmo arquivo, implementaremos o teste abaixo que faz uma requisição para rota "/**sair**", e seu comportamento de sucesso é receber um redirecionamento para rota principal "/", que será o retorno da variável: `res.headers.location`.

```
it('deve ir para rota / ao fazer GET /sair', function(done){
  request.get('/sair')
         .end(function(err, res){
    res.headers.location.should.eql('/');
    done();
  });
});
```

Aqui já complicamos o teste, simulamos um POST enviando parâmetros válidos `(nome e email)`, que faz um login e é redirecionado para rota "/**contatos**".

```
it('deve ir para rota /contatos ao fazer POST /entrar', function(done){
  var login = {usuario: {nome: 'Teste', email: 'teste@teste'}};
  request.post('/entrar')
         .send(login)
         .end(function(err, res){
    res.headers.location.should.eql('/contatos');
    done();
  });
});
```

Este último teste, é semelhante ao anterior, a diferença é que enviamos parâmetros inválidos de login, para que seja testado o comportamento de redirecionamento da rota.

```
it('deve ir para rota / ao fazer POST /entrar', function(done){
    var login = {usuario: {nome: '', email: ''}};
    request.post('/entrar')
           .send(login)
           .end(function(err, res){
      res.headers.location.should.eql('/');
      done();
    });
  });
}); // fim da função describe()
```

Esse foi o nosso teste com a rota `home.js`. Cada teste deixei bem descritivo e atômico, realizando apenas uma única verificação através da função `should`. Vamos rodar os testes para ver se tudo deu certo?

Para executar os testes, é simples! Como já configuramos no `package.json` o script para execução de testes, execute no terminal o comando `npm test`. E veja o resultado dos seus testes semelhantes ao da imagem abaixo:

Figura 8.2: Os testes em home.js passaram com sucesso.

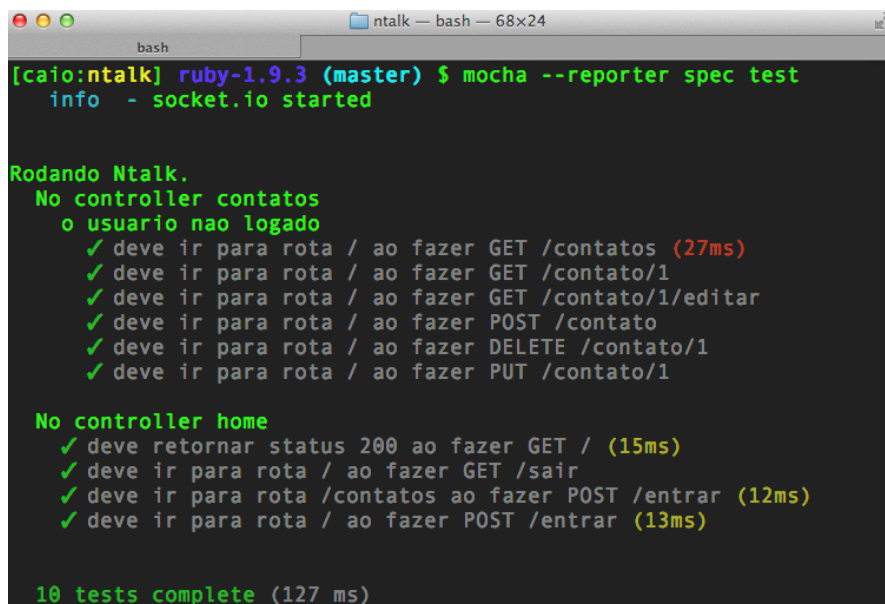
Reparem que surgiram dois prints do sistema que só aparecem quando levantamos o servidor, e em seguida aparece o resultado dos testes. Isso acontece porque em `app.js` exportamos a variável que contém as funções do servidor da aplicação, o `module.exports = app`. Nos testes, quando carregamos este módulo e injetamos dentro do `require('supertest')(app)`, ele se encarrega de iniciar o servidor para que o `supertest` tenha as informações necessárias para emular requisições e assim permitir que façamos os testes funcionais das rotas.

Agora iremos explorar novas funções do Mocha, testando as rotas do controller `contatos.js`. Como este controller possui um filtro que verifica se existe um usuário logado no sistema, implementaremos dois casos de testes: um caso de testes para usuário logado e um caso para um usuário não logado.

Crie o arquivo `test/contatos.js`. Nele vamos criar um `describes` com dois `sub-describes` que serão utilizados para criar os casos de testes para usuário logado e não logado.

```
var app = require('../app')
  , should = require('should')
  , request = require('supertest')(app);

describe('No controller contatos', function() {

  describe('o usuario nao logado', function() {
    // testes aqui...
  });

  describe('o usuario logado', function() {
    // testes aqui...
  });

});
```

No caso de testes para usuário não logado implementaremos os simples testes de verificações, semelhante ao que utilizamos no `test/home.js`. Afinal, em `routes/contatos.js`, existe um filtro que irá barrar um usuário não logado, fazendo com que a requisição seja redirecionada para a rota principal: '/'.

Para simplificar, seguem todos os testes que serão inseridos dentro de `describe('o usuario nao logado')`:

```
it('deve ir para / ao fazer GET /contatos', function(done){
  request.get('/contatos').end(function(err, res) {
    res.headers.location.should.eql('/');
    done();
  });
});
it('deve ir para / ao fazer GET /contato/1', function(done){
  request.get('/contato/1').end(function(err, res) {
    res.headers.location.should.eql('/');
```

```
    done();
  });
});
it('deve ir para / ao fazer GET /contato/1/editar', function(done){
  request.get('/contato/1/editar').end(function(err, res) {
    res.headers.location.should.eql('/');
    done();
  });
});
it('deve ir para / ao fazer POST /contato', function(done){
  request.post('/contato').end(function(err, res) {
    res.headers.location.should.eql('/');
    done();
  });
});
it('deve ir para / ao fazer DELETE /contato/1', function(done){
  request.del('/contato/1').end(function(err, res) {
    res.headers.location.should.eql('/');
    done();
  });
});
it('deve ir para / ao fazer PUT /contato/1', function(done){
  request.put('/contato/1').end(function(err, res) {
    res.headers.location.should.eql('/');
    done();
  });
});
```

Reparem que o teste dentro de `describe('o usuario nao logado')` foi muito repetitivo. Isso ocorre pois o filtro que aplicamos no **capítulo 4** vai barrar qualquer requisição de usuário não autenticado pelo login, por isso o resultado correto é que todos os testes redirecionem para a rota principal do sistema, que é a tela de login.

Mais uma vez vamos rodar os testes para ter a certeza de que tudo ocorreu bem na implementação dos testes. Dessa vez vamos ver em mais detalhes os resultados. Para isso, execute o comando `mocha test --reporter spec`, e os resultados serão apresentados no formato semelhante ao framework RSpec da linguagem Ruby, igual à imagem abaixo:

```
ntalk — bash — 68×24
bash
[caio:ntalk] ruby-1.9.3 (master) $ mocha --reporter spec test
   info  - socket.io started

Rodando Ntalk.
  No controller contatos
    o usuario nao logado
      ✓ deve ir para rota / ao fazer GET /contatos (27ms)
      ✓ deve ir para rota / ao fazer GET /contato/1
      ✓ deve ir para rota / ao fazer GET /contato/1/editar
      ✓ deve ir para rota / ao fazer POST /contato
      ✓ deve ir para rota / ao fazer DELETE /contato/1
      ✓ deve ir para rota / ao fazer PUT /contato/1

  No controller home
    ✓ deve retornar status 200 ao fazer GET / (15ms)
    ✓ deve ir para rota / ao fazer GET /sair
    ✓ deve ir para rota /contatos ao fazer POST /entrar (12ms)
    ✓ deve ir para rota / ao fazer POST /entrar (13ms)


  10 tests complete (127 ms)
```

Figura 8.3: Resultado dos testes utilizando Reporter Spec.

O mais legal do Mocha é que ele possui diversos *reporters*, permitindo que você tenha várias opções para customizar o resultado de seus testes. Para conhecer outros formatos de *reporters* veja este link:

http://visionmedia.github.io/mocha/#reporters

No `describe('o usuario logado')`, teremos que emular um usuário autenticado, ou seja, um usuário que fez um login no sistema. Para isso utilizaremos duas estratégias:

1) Cada teste deve, **antes**, fazer uma requisição `POST` na rota de login.

2) O mesmo teste deve manter um **cookie** válido, gerado após o login obter sucesso, para pular o filtro.

Para implementar essa rotina, utilizaremos a função `beforeEach()`, que é executada antes de cada teste. Dentro dessa função, faremos um login no sistema para capturar o seu `cookie`, que é encontrado dentro de `res.headers['set-cookie']`. Nisso, armazenaremos seu resultado em uma variável que estará no mesmo escopo da função `describe('o usuario`

logado') para que seja reutilizado em cada um de seus testes. Para entender melhor, veja o código abaixo:

```
describe('o usuario logado', function() {
  var login = {usuario: {nome: 'Teste', email: 'teste@teste'}}
    , contato = {contato: {nome: 'Teste', email: 'teste@teste'}}
    , cookie = {};

  beforeEach(function(done) {
    request.post('/entrar')
           .send(login)
           .end(function(err, res) {
      cookie = res.headers['set-cookie'];
      done();
    });
  });
  // implementação dos testes...
});
```

Com a variável `cookie` recebendo os dados de um usuário autenticado, fica viável testar o comportamento das rotas pós-filtro. Para executar os testes, temos que injetar o `cookie` em cada requisição, e isso se faz via `req.cookies = cookie`. Vamos implementá-los? Abaixo utilizaremos esta técnica, mas infelizmente testaremos apenas algumas rotas. Após a apresentação dos testes explicarei o motivo.

Aqui testamos uma requisição GET na rota **/contatos**:

```
// função beforeEach()...
it('deve retornar status 200 em GET /contatos', function(done){
  var req = request.get('/contatos');
  req.cookies = cookie;
  req.end(function(err, res) {
    res.status.should.eql(200);
    done();
  });
});
```

No teste abaixo, emulamos uma requisição POST na rota **/contato**, testando o comportamento do cadastro de um contato:

```
it('deve ir para rota /contatos em POST /contato', function(done){
    var contato = {contato: {nome: 'Teste', email: 'teste@teste'}};
```

```
    var req = request.post('/contato');
    req.cookies = cookie;
    req.send(contato).end(function(err, res) {
      res.headers.location.should.eql('/contatos');
      done();
    });
  });
}); // fim da função describe()
```

E então vem a pergunta, porque não foram testadas todas as rotas para um usuário logado? Testes de aceitação testam o comportamento do sistema simulando uma requisição real, pelo qual testamos as rotas. Este tipo de teste aqui está automatizado, mas por ele ser um típico *teste de caixa-preta*, também é possível realizá-lo manualmente, como um usuário acessando o sistema. As rotas de `contatos.js`, que passam um `id` como parâmetro, precisam de um `id` válido que retorne um objeto do banco de dados, e como testamos somente o retorno das requisições, não há a possibilidade de criar objetos *fakes*, impossibilitando a elaboração destes testes.

8.6 Deixando seus testes mais limpos

Algo que polui muito os códigos de teste são as variáveis principais, que são carregadas no topo. É claro que por questões de legibilidade e entendimento do teste é necessário declará-los em cada teste, porém é possível criar um arquivo de configuração do próprio Mocha para que sejam centralizados em um único arquivo os parâmetros iniciais de execução do Mocha e seus módulos auxiliares. Este arquivo deve ser incluído dentro do diretório `test`, com o nome `mocha.opts`. Basicamente ele permite utilizar os parâmetros de configuração do seu próprio CLI, assim como também permite carregar alguns módulos auxiliares: `should`. E isso será o suficiente para deixar os testes mais limpos.

Dentro de `test`, crie o arquivo `mocha.opts`, seguindo os parâmetros do código abaixo:

```
--require should
--report spec
```

Além de carregarmos o `should`, também foi adicionado um novo parâmetro, o `--report spec`, que define o layout do resultado dos testes. Caso queria incluir outros parâmetros em seu `mocha.opts`, execute no terminal o comando `mocha -h` para visualizar todas opções de configuração.

Figura 8.4: Parâmetros opcionais do Mocha.

Agora, só para finalizar, remova a função `var should = require('should')` de todos os testes, pois eles serão automaticamente carregados via `mocha.opts`.

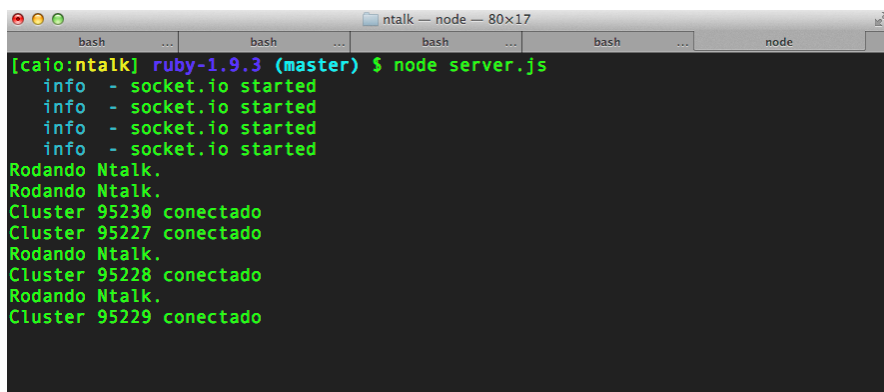
Capítulo 9

Aplicação Node em produção

9.1 O que vamos fazer?

Enfim, chegamos no último capítulo desse livro! Nas próximas seções serão abordados temas importantes para preparar o nosso projeto para o ambiente de produção. O objetivo aqui é apresentar alguns conceitos e ferramentas para manter uma aplicação Node de forma segura e com boa performance. Otimizaremos o projeto Ntalk, preparando-o para entrar em ambiente de produção, além de garantir toda monitoria do sistema através de *loggings*.

9.2 Configurando Clusters

Infelizmente, o Node.js não trabalha com *threads*, isso é algo que na opinião de alguns desenvolvedores é considerado como um ponto negativo, pelo qual desperta um certo desinteresse em aprender ou levar a sério esta tecnologia. Mas apesar do Node ser *single-thread* é possível, sim, prepará-lo para trabalhar com processamento paralelo. Para isso existe nativamente um módulo chamado de `Cluster`.

Ele basicamente instancia novos processos de uma aplicação, trabalhando de forma distribuída e compartilhando a mesma porta da rede. O número de processos a serem criados quem determina é você, e é claro que a boa prática é instanciar um total de processos relativo à quantidade de núcleos do processador do servidor. Por exemplo, se tenho um processador de oito núcleos, então posso instanciar oito processos, criando assim uma rede de oito *clusters*.

Para garantir que os *clusters* trabalhem de forma distribuída e organizada é necessário que exista um processo pai, mais conhecido como *cluster master*. Ele é o processo responsável por balancear a carga de processamento, distribuindo entre os demais processos que são chamados de *cluster slave*. Implementar essa técnica no Node.js é muito simples, visto que toda distribuição entre os *clusters* é executada de forma abstraída para o desenvolvedor. Outra vantagem é que os *clusters* são independentes uns dos outros, ou seja, caso um *cluster* saia do ar, os demais continuarão servindo a aplicação mantendo o sistema no ar. Porém, é necessário gerenciar as instancias e encerramento desses processos manualmente. Com base nesses conceitos, vamos aplicar na prática a implementação de *clusters*. Crie no diretório raiz o arquivo `clusters.js`, para que através dele seja carregado clusters da nossa aplicação, veja o código abaixo:

```
var cluster = require('cluster')
  , cpus = require('os').cpus()
;
if (cluster.isMaster) {
  cpus.forEach(function(cpu) {
    cluster.fork();
  });
  cluster.on('listening', function(worker) {
    console.log("Cluster %d conectado", worker.process.pid);
  });
  cluster.on('disconnect', function(worker) {
    console.log('Cluster %d esta desconectado.', worker.process.pid);
  });
  cluster.on('exit', function(worker) {
    console.log('Cluster %d caiu fora.', worker.process.pid);
  });
} else {
  require('./app');
}
```

Dessa vez, para levantar o servidor será via comando `node clusters.js` para

que a aplicação rode de forma distribuída e para comprovar que deu certo. Veja no terminal quantas vezes se repetiu a mensagem: `"Ntalk no ar"`.

```
ntalk — node — 80×17
bash   bash   bash   bash   node
[caio:ntalk] ruby-1.9.3 (master) $ node server.js
   info  - socket.io started
   info  - socket.io started
   info  - socket.io started
   info  - socket.io started
Rodando Ntalk.
Rodando Ntalk.
Cluster 95230 conectado
Cluster 95227 conectado
Rodando Ntalk.
Cluster 95228 conectado
Rodando Ntalk.
Cluster 95229 conectado
```

Figura 9.1: Rodando Node.js em clusters.

Basicamente, carregamos o módulo `cluster` e primeiro verificamos se ele é o *cluster master* via função `cluster.isMaster`. Caso ele seja, rodamos um loop cuja suas iterações é baseada no total de *cpus* que ocorre através do trecho `cpus.forEach()`, que retorna o total de núcleos do servidor. Em cada iteração rodamos o `cluster.fork()` que, na prática, instancia um `child process` (processo filho) desta aplicação. Quando nasce um novo processo (neste caso um processo-filho), consequentemente ele não cai na condicional: `if(cluster.isMaster)`. Com isso é iniciado o servidor da aplicação via `require('./app')` através de um *cluster slave*.

Também foram incluídos alguns eventos emitidos pelo *cluster master*, no código existem apenas os principais eventos:

- `listening`: acontece quando um *cluster* está escutando uma porta do servidor. Neste caso, a nossa aplicação está escutando a porta 4000.
- `disconnect`: executa seu callback quando um *cluster* se desconecta da rede.
- `exit`: ocorre quando um processo-filho é fechado no sistema operacional.

Desenvolvimento em Clusters

Muito pode ser explorado no desenvolvimento de *clusters* no Node.js. Aqui apenas aplicamos o essencial para manter nossa aplicação rodando em paralelo, mas caso tenha a necessidade de implementar mais detalhes que explorem ao máximo os *clusters*, recomendo que leia a documentação — (http://nodejs.org/api/cluster.html) — para ficar por dentro de todos os eventos e funções deste módulo.

Para finalizar e deixar automatizado o *start* do servidor em modo *cluster* via comando `npm`, atualize em seu `package.json` no atributo `scripts` de acordo com o código abaixo:

```
"scripts": {
  "start": "node clusters",
  "test": "NODE_ENV=test ./node_modules/mocha/bin/mocha test/*.js"
}
```

Pronto! Agora você pode executar sua aplicação através do comando `npm start`.

9.3 Redis controlando as sessões da aplicação

Quando desenvolvemos no Node.js uma aplicação orientada a clusters, aliado ao framework Express e Socket.IO, seus mecanismos default de persistência de Sessions param de funcionar corretamente. Não acredita? Então veja você mesmo! Execute o servidor via `npm start`, agora faça um login no sistema, até agora esta tudo ok, correto? Tente cadastrar um novo contato ou ver os detalhes de um existente. Repare que automaticamente você foi redirecionado para tela de login, mas que estranho! Por que aconteceu isso? No momento utilizamos um controle de sessão em memória (conhecido pelo nome: `MemoryStore`). A natureza desse tipo de controle não consegue compartilhar dados entre os clusters, pois ele foi projetado para trabalhar com apenas um processo. O Express e Socket.IO são frameworks que utilizam por padrão sessão em memória. A solução para este problema é adotar um novo tipo de *store* para a sessão, sendo o Redis uma ótima alternativa. Como já o utilizamos dentro do chat da aplicação teremos agora apenas que adaptá-lo para o mecanismo *session store* do Express e do Socket.IO.

Essa adaptação é simples, e seu resultado visa manter a aplicação rodando perfeitamente em clusters. Implantaremos esse upgrade no mecanismo de sessão do Express e Socket.IO. Primeiro criaremos uma nova *lib* focado em gerenciar conexões do Redis, este terá funções para retornar um simples cliente Redis, um *Redis Store* para o Express e um *Redis Store* para o Socket.IO. Crie o arquivo `lib/redis_connect.js` e implemente o código abaixo:

```
var redis = require('redis')
  , redisStore = require('connect-redis')
  , express = require('express')
  , socketio = require('socket.io')
;

exports.getClient = function() {
  return redis.createClient();
}
exports.getExpressStore = function() {
  return redisStore(express);
}
exports.getSocketStore = function() {
  return socketio.RedisStore;
}
```

Agora para que este código funcione execute no terminal o comando: `npm install connect-redis --save`. E para finalizar faremos o *upgrade* das *stores*, através do *refactoring* no código `app.js`:

```
var express = require('express')
  , app = express()
  , load = require('express-load')
  , server = require('http').createServer(app)
  , error = require('./middleware/error')
  , io = require('socket.io').listen(server)
  , redis = require('./lib/redis_connect')
  , ExpressStore = redis.getExpressStore()
  , SocketStore = redis.getSocketStore()
;
const SECRET = 'Ntalk', KEY = 'ntalk.sid';
var cookie = express.cookieParser(SECRET)
  , storeOpts = {client: redis.getClient(), prefix: KEY}
  , store = new ExpressStore(storeOpts)
```

```
  , sessOpts = {secret: SECRET, key: KEY, store: store}
  , session = express.session(sessOpts);

// stack de configurações do Express...
io.set('store', new SocketStore);
// io.set('authorization')...
// load()...
// server.listen()...
```

Também modificaremos o `sockets/chat.js` para que ele receba uma conexão Redis diretamente dessa *lib*:

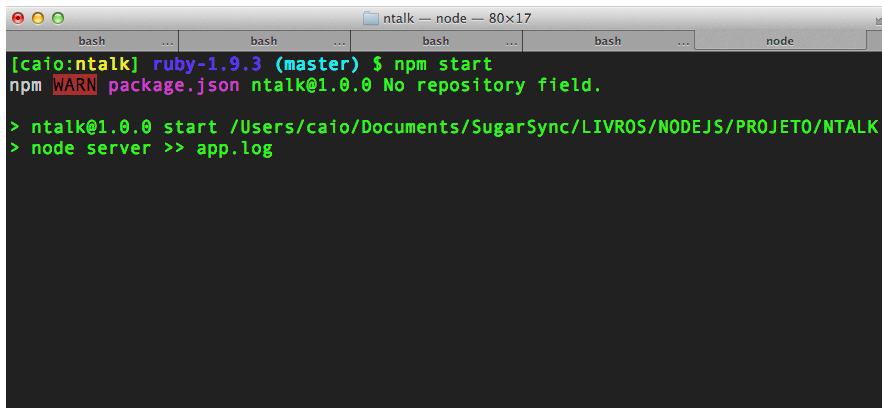
```
module.exports = function(io) {

  var crypto = require('crypto')
    , redis_connect = require('../lib/redis_connect')
    , redis = redis_connect.getClient()
    , sockets = io.sockets;

  // continuação dos eventos do socket.io...
}
```

Com esse upgrade implementado agora o sistema vai rodar perfeitamente em clusters, sem causar bugs no controle de sessão e todas as sessões serão compartilhadas entre os clusters existentes.

9.4 Monitorando aplicação através de logs

Quando colocamos um sistema em produção, aumentamos os riscos de acontecerem bugs que não foram identificados durante os testes no desenvolvimento. Isso é normal, e toda aplicação já passou ou vai passar por esta situação. O importante neste caso é ter um meio de monitorar todo comportamento da aplicação, através de arquivos de *logs*. Tanto o Express quanto o Socket.IO possuem um *middleware* para monitorar e gerar *logs*. Sua instalação é simples, abra o `app.js` e adicione no topo da *stack* de configurações do Express a função `app.use(express.logger())`:

```
app.use(express.logger());
app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
app.use(cookie);
```

```
app.use(session);
app.use(express.json());
app.use(express.urlencoded());
app.use(express.methodOverride());
app.use(app.router);
app.use(express.static(__dirname + '/public'));
app.use(error.notFound);
app.use(error.serverError);
```

Para configurar os logs no Socket.IO é simples também! Adicione no `app.js` a função `io.set('log level', 1)` antes da função `io.set('authorization')`.

```
io.set('log level', 1);
```

Agora o seu sistema está gerando logs com maior detalhe de todo comportamento da aplicação. O único problema aqui é que estes logs serão impressos na tela de console. Em um sistema em produção, seria muito tedioso ficar com console do sistema aberto para ver os logs, afinal você também tem uma vida no mundo real para viver! Se do nada acontecer um problema no sistema e você não estiver presente para ver o erro gerado no console, você perderá informações úteis de debug da aplicação. Para resolver este problema, é necessário um simples comando no terminal. Ao executar o comando `node clusters >> app.log` o terminal para de imprimir logs na tela e passa escrever os logs dentro do arquivo `app.log`. E para simplificar ainda mais, vamos manter este comando dentro do *alias* `npm start`. Abra o `package.json` e altere a seguinte linha:

```
"scripts": {
  "start": "node clusters >> app.log",
  "test": "NODE_ENV=test ./node_modules/mocha/bin/mocha test/*.js"
}
```

Agora sua aplicação esta preparada para gerar logs em arquivo de texto. Para testar as alterações execute o comando `npm start`. Repare que desta vez o terminal vai ficar congelado sem exibir nenhuma mensagem na tela, veja a imagem abaixo:

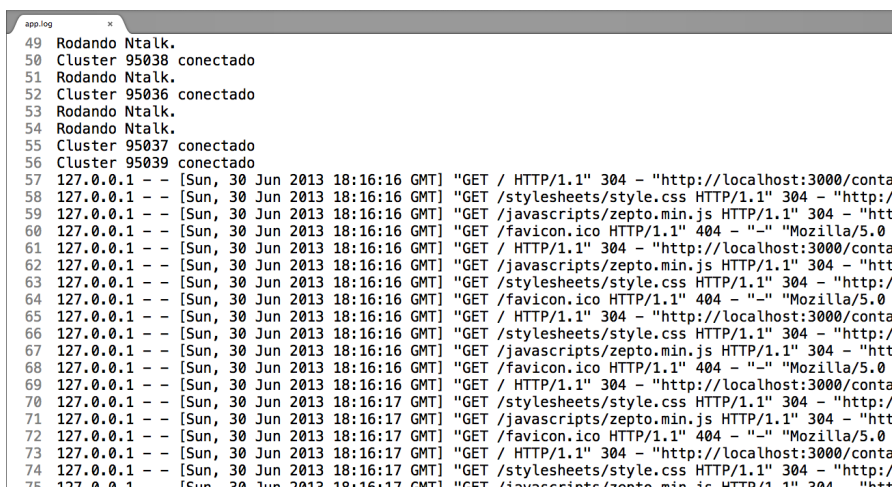



Figura 9.2: Tela do terminal não emitindo logs.

Em contrapartida, todas as mensagens serão persistidas dentro do arquivo `app.log`.

```
app.log
49 Rodando Ntalk.
50 Cluster 95038 conectado
51 Rodando Ntalk.
52 Cluster 95036 conectado
53 Rodando Ntalk.
54 Rodando Ntalk.
55 Cluster 95037 conectado
56 Cluster 95039 conectado
57 127.0.0.1 - - [Sun, 30 Jun 2013 18:16:16 GMT] "GET / HTTP/1.1" 304 - "http://localhost:3000/conta
58 127.0.0.1 - - [Sun, 30 Jun 2013 18:16:16 GMT] "GET /stylesheets/style.css HTTP/1.1" 304 - "http:/
59 127.0.0.1 - - [Sun, 30 Jun 2013 18:16:16 GMT] "GET /javascripts/zepto.min.js HTTP/1.1" 304 - "htt
60 127.0.0.1 - - [Sun, 30 Jun 2013 18:16:16 GMT] "GET /favicon.ico HTTP/1.1" 404 - "-" "Mozilla/5.0
61 127.0.0.1 - - [Sun, 30 Jun 2013 18:16:16 GMT] "GET / HTTP/1.1" 304 - "http://localhost:3000/conta
62 127.0.0.1 - - [Sun, 30 Jun 2013 18:16:16 GMT] "GET /javascripts/zepto.min.js HTTP/1.1" 304 - "htt
63 127.0.0.1 - - [Sun, 30 Jun 2013 18:16:16 GMT] "GET /stylesheets/style.css HTTP/1.1" 304 - "http:/
64 127.0.0.1 - - [Sun, 30 Jun 2013 18:16:16 GMT] "GET /favicon.ico HTTP/1.1" 404 - "-" "Mozilla/5.0
65 127.0.0.1 - - [Sun, 30 Jun 2013 18:16:16 GMT] "GET / HTTP/1.1" 304 - "http://localhost:3000/conta
66 127.0.0.1 - - [Sun, 30 Jun 2013 18:16:16 GMT] "GET /stylesheets/style.css HTTP/1.1" 304 - "http:/
67 127.0.0.1 - - [Sun, 30 Jun 2013 18:16:16 GMT] "GET /javascripts/zepto.min.js HTTP/1.1" 304 - "htt
68 127.0.0.1 - - [Sun, 30 Jun 2013 18:16:16 GMT] "GET /favicon.ico HTTP/1.1" 404 - "-" "Mozilla/5.0
69 127.0.0.1 - - [Sun, 30 Jun 2013 18:16:16 GMT] "GET / HTTP/1.1" 304 - "http://localhost:3000/conta
70 127.0.0.1 - - [Sun, 30 Jun 2013 18:16:17 GMT] "GET /stylesheets/style.css HTTP/1.1" 304 - "http:/
71 127.0.0.1 - - [Sun, 30 Jun 2013 18:16:17 GMT] "GET /javascripts/zepto.min.js HTTP/1.1" 304 - "htt
72 127.0.0.1 - - [Sun, 30 Jun 2013 18:16:17 GMT] "GET /favicon.ico HTTP/1.1" 404 - "-" "Mozilla/5.0
73 127.0.0.1 - - [Sun, 30 Jun 2013 18:16:17 GMT] "GET / HTTP/1.1" 304 - "http://localhost:3000/conta
74 127.0.0.1 - - [Sun, 30 Jun 2013 18:16:17 GMT] "GET /stylesheets/style.css HTTP/1.1" 304 - "http:/
```

Figura 9.3: Logs da aplicação no arquivo app.log.

9.5 Otimizações no Express

Nesta seção pretendo passar algumas dicas que visam aumentar a performance do sistema. Serão adicionadas algumas configurações tanto para o Express como para o

Socket.IO e Mongoose, com o objetivo de otimizar tanto server-side como o client-side da aplicação.

Toda otimização será feita dentro do `app.js`, afinal ele faz o *boot* da nossa aplicação, pelo qual ele carrega e executa todos seus submódulos. Vamos começar otimizando o Express. Faremos nele 2 otimizações: habilitar compactação gzip através do novo stack `express.compress()` e adicionaremos cache para os arquivos estáticos incluindo dentro de `express.static` o atributo `maxAge`. Veja abaixo como será essas alterações no `app.js`:

```
const SECRET = 'Ntalk', KEY = 'ntalk.sid'
    , MAX_AGE = {maxAge: 3600000}
    , GZIP_LVL = {level: 9, memLevel: 9};

// configurações de session e cookies...

app.use(express.logger('dev'));
app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
app.use(cookie);
app.use(session);
app.use(express.json());
app.use(express.urlencoded());
app.use(express.methodOverride());
app.use(express.compress(GZIP_LVL));
app.use(app.router);
app.use(express.static(__dirname + '/public', MAX_AGE));
app.use(error.notFound);
app.use(error.serverError);
```

9.6 Otimizando requisições do Socket.IO

Já no Socket.IO, habilitaremos `minification + cache + gzip + etag` para otimizar as requisições e mudaremos seu nível de logs para informar apenas quando ocorrer erros na aplicação:

```
io.enable('browser client cache');
io.enable('browser client minification');
io.enable('browser client etag');
io.enable('browser client gzip');
io.set('log level', 1);
```

```
io.set('store', new SocketStore);
/// io.set('authorization') ...
```

Para deixar mais *clean* nosso `app.js`, podemos deixar externo essas configurações, chamando-as via comando `require` que além de carregar módulos, também carrega arquivos `.json`. Para isso crie na raiz do projeto o arquivo `config.json` com os itens abaixo:

```
{
  "SECRET": "Ntalk",
  "KEY": "ntalk.sid",
  "MAX_AGE": {
    "maxAge": 3600000
  },
  "GZIP_LVL": {
    "level": 9,
    "memLevel": 9
  }
}
```

Agora voltando em `app.js` carregaremos o `config.json` via comando `var cfg = require('./config.json')`:

```
var express = require('express')
  , app = express()
  , load = require('express-load')
  , server = require('http').createServer(app)
  , error = require('./middleware/error')
  , cfg = require('./config.json')
  , io = require('socket.io').listen(server)
  , redis = require('./lib/redis_connect')
  , ExpressStore = redis.getExpressStore()
  , SocketStore = redis.getSocketStore()
;
```

Atualizaremos as constantes atuais para serem chamadas pela variável `cfg`, primeiro modificaremos as configurações de cookie e sessions:

```
var cookie = express.cookieParser(cfg.SECRET)
  , storeOpts = {client: redis.getClient(),
                 prefix: cfg.KEY}
  , store = new ExpressStore(storeOpts)
```

```
  , sessOpts = {secret: cfg.SECRET,
                key: cfg.KEY,
                store: store}
  , session = express.session(sessOpts)
;
```

E para finalizar atualizaremos as configurações de *middlewares* do Express e Socket.IO:

```
app.use(express.logger('dev'));
app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
app.use(cookie);
app.use(session);
app.use(express.json());
app.use(express.urlencoded());
app.use(express.methodOverride());
app.use(express.compress(cfg.GZIP_LVL));
app.use(app.router);
app.use(express.static(__dirname + '/public', cfg.MAX_AGE));
app.use(error.notFound);
app.use(error.serverError);

io.enable('browser client cache');
io.enable('browser client minification');
io.enable('browser client etag');
io.enable('browser client gzip');
io.set('log level', 1);
io.set('store', new SocketStore);
io.set('authorization', function(data, accept) {
  cookie(data, {}, function(err) {
    var sessionID = data.signedCookies[cfg.KEY];
    store.get(sessionID, function(err, session) {
      if (err || !session) {
        accept(null, false);
      } else {
        data.session = session;
        accept(null, true);
      }
    });
  });
});
```

9.7 Aplicando Singleton nas conexões do Mongoose

No Mongoose, apenas aplicaremos o *design pattern Singleton* para instanciar as conexões do banco de dados. Isso será implementado com o objetivo de garantir que apenas uma única conexão seja instanciada e compartilhada por toda aplicação. No arquivo `lib/db_connect.js`, codifique aplicando as seguintes mudanças:

```
var mongoose = require('mongoose')
  , single_connection
  , env_url = {
    "test": "mongodb://localhost/ntalk_test",
    "development": "mongodb://localhost/ntalk"
  }
;
module.exports = function() {
  var env = process.env.NODE_ENV || "development"
    , url = env_url[env];
  if(!single_connection) {
    single_connection = mongoose.connect(url);
  }
  return single_connection;
};
```

9.8 Mantendo o sistema no ar com Forever

O Node.js é praticamente uma plataforma de baixo nível, com ele temos bibliotecas com acesso direto aos recursos do sistema operacional, e programamos entre diversos protocolos, como por exemplo, o protocolo http. Para trabalhar com http, temos que programar como será o servidor http e também sua aplicação. Quando colocamos uma aplicação Node em produção diversos problemas e bugs são encontrados com o passar do tempo, e quando surge um bug grave o servidor cai, deixando a aplicação fora do ar. De fato, programar em Node.js requer lidar com esses detalhes de servidor.

O Forever é uma ferramenta que surgiu para resolver esse problema de queda do servidor. Seu objetivo é monitorar um servidor realizando *pings* a cada curto período predeterminado pelo desenvolvedor. Quando ele detecta que a aplicação está fora do ar, automaticamente ele dá um restart nela. Ele consegue fazer isso por que mantém a aplicação rodando em *background* no sistema operacional.

Existem duas versões deste framework, uma é a versão CLI em que toda tarefa é

realizada no terminal. A outra maneira de se trabalhar com o `forever` é de forma programável em código Node, utilizando o módulo `forever-monitor`. Este último, é o mais recomendado a se utilizar quando sua aplicação esta hospedada em um ambiente cujo terminal possui restrições de segurança que não permite instalar programas do tipo CLI. E o mecanismo do `forever-monitor` é o mesmo que o `forever`. Sua única diferença é que sua configuração é feita via Javascript, e ele instancia um aplicação Node via `child process`.

Sua instalação é simples, basta executar o comando: `npm install forever-monitor --save`, para instalá-lo e auto atualizar o seu `package.json`.

Agora vamos criar um novo código chamado de `server.js` dentro do diretório raiz do projeto. Algo interessante do `forever-monitor` é que, além de ele reiniciar o servidor, ele gera arquivos de logs da aplicação separados por logs do forever (`logFile`), logs da aplicação (`outFile`) e logs de erros da aplicação (`errFile`). Veja abaixo mais detalhes de como será este código-fonte:

```
var forever = require('forever-monitor');
var Monitor = forever.Monitor;

var child = new Monitor('clusters.js', {
  max: 10,
  silent: true,
  killTree: true,
  logFile: 'forever.log',
  outFile: 'app.log',
  errFile: 'error.log'
});

child.on('exit', function () {
  console.log('O servidor foi finalizado.');
});

child.start();
```

Tudo começa quando instanciamos o objeto `Monitor`. Nele passamos dois parâmetros em seu construtor, o primeiro é o código da aplicação que desejamos executar (no nosso caso é o `cluster.js`) e no segundo parâmetro passamos um objeto com os atributos de configuração do `forever-monitor`. Em `max` definimos o total de vezes que poderá reiniciar o servidor quando ele cair, lembrando que, ao

passar deste total, sua aplicação será totalmente finalizada. Infelizmente esta é uma limitação do `forever-monitor`, pois o `forever` por ser um CLI, permite reiniciar infinitamente sua aplicação. O atributo `silent` apenas oculta a exibição de logs no terminal. Ao habilitar o atributo: `killTree`, todos os processos-filhos da sua aplicação serão finalizados a cada restart do servidor.

Agora que temos o `forever-monitor` configurado e gerando logs por conta própria, vamos atualizar o comando `npm start` para que ele execute diretamente o `server.js` ao invés do atual `clusters.js`. Com base no código abaixo, edite o seu `package.json`:

```
"scripts": {
  "start": "node server",
  "test": "NODE_ENV=test ./node_modules/mocha/bin/mocha test/*.js"
}
```

Esta foi uma configuração básica para utilizar o `forever-monitor`. Caso sua necessidade seja ir além do que foi apresentado aqui, visite o github oficial dos projetos (https://github.com/nodejitsu/forever) e (https://github.com/nodejitsu/forever-monitor) . Cada um possui suas vantagens e desvantagens, basta saber qual alternativa terá melhor resultado de acordo com o ambiente que será hospedado sua aplicação Node.js. Lembrando que em ambientes limitados que não permitem instalar CLIs a melhor alternativa é usar o `forever-monitor`.

Agora, nossa aplicação está otimizada para o ambiente de produção. Na próxima seção faremos uma integração interessante com o servidor Nginx, que é considerado como um ótimo servidor de arquivos estáticos.

9.9 Integrando Nginx no Node.js

Enfim, estamos na última seção. Durante todo o percurso implementamos uma aplicação Node.js que utiliza o web framework Express, persiste dados tanto para o MongoDB, como para o Redis e realiza comunicação bidirecional através do Socket.IO. Também configuramos clusters, logs e codificamos testes de aceitação utilizando o Mocha + Supertest. Em especial tivemos dois capítulos dedicados ao Express, afinal ele é a base principal da nossa aplicação, com ele desenvolvemos rotas e incluímos diversas stacks que visam otimizar o fluxo do servidor http. Nesta seção iremos integrar o Node.js com o servidor Nginx.

Figura 9.4: Servidor Nginx.

O objetivo dessa integração visa aumentar performance da aplicação pelo qual criaremos um *proxy* do Nginx com Node.js e também delegaremos todo processamento de arquivos estático para o Nginx, deixando apenas que o Node.js cuide do processamento de suas rotas. Isso diminui o número de requisições diretas em nossa aplicação.

Atenção: Não entraremos em detalhes sobre como instalar o Nginx em sua máquina. Para instalá-lo recomendo que acesse seu site oficial: (http://nginx.org) . Também recomendo que leia sua *Wiki* que contém diversas dicas de como configurá-lo: (http://wiki.nginx.org/Main) . A versão utilizada neste livro é a versão `1.5.2`, recomendo que **não** utilize versões anteriores a esta, pois é provável que não funcione a dica de configuração que explicarei abaixo. Outro detalhe importante é que foi a partir da versão **1.3.13** que o Nginx passou a dar suporte ao protocolo WebSocket, e nisso o seu servidor estará habilitado para processar o WebSocket do Socket.IO em browsers compatíveis.

Agora que temos o Nginx instalado e funcionando corretamente em sua máquina, vamos configurá-lo para que ele comece a servir arquivos estáticos de nossa aplicação, tudo isso será feito dentro de seu arquivo principal chamado `nginx.conf`. A localização deste arquivo varia de acordo com o sistema operacional, então recomendo que leia sua documentação oficial: (http://nginx.org/en/docs) para descobrir onde ele se encontra.

Abaixo apresento uma versão simplificada de configuração do Nginx. Esta configuração fará o Nginx servir os arquivos estáticos ao invés do Express, e para finalizar aplicamos um *proxy* do Nginx para as rotas da nossa aplicação.

```
worker_processes 1;

events {
  worker_connections 1024;
}

http {
  include mime.types;
  default_type application/octet-stream;
  sendfile on;
  keepalive_timeout 65;
  gzip on;

  server {
    listen 80;
    server_name localhost;
    access_log logs/access.log;

    location ~ ^/(javascripts|stylesheets|images) {
      root /var/www/ntalk/public;
      expires max;
    }

    location / {
      proxy_pass http://localhost:3000;
    }
  }
}
```

Praticamente adicionamos algumas melhorias em cima das configurações padrões do `nginx.conf`. Com o objetivo de otimizar o servidor estático, habilitamos compactação *gzip* nativa, através do trecho: `gzip on;` e criamos dois `locations` dentro de `server`. O primeiro `location` é o responsável por servir conteúdo estático.

```
location ~ ^/(javascripts|stylesheets|images) {
  root /var/www/ntalk/public;
```

```
  expires max;
}
```

É dentro dele definimos a localização da pasta `public` da nossa aplicação através do trecho: `root /var/www/ntalk/public;`. Esta localização definida no item `root` se baseia no endereço onde fica a pasta `public` do projeto em seu sistema operacional, seja ele Unix, MacOS ou Linux. Se o seu sistema é Windows altere o endereço para o padrão de diretórios dele, que é algo semelhante a `root C:/www/ntalk/public` ou para qualquer outro endereço que você deseja manter este projeto. Também aplicamos dentro desse `location` um *cache* dos arquivos através do item: `expires max;`.

No último `location`, aplicamos um simples controle de *proxy*. O item `proxy_pass` praticamente redireciona as demais rotas para nossa aplicação, que estará ativa através do endereço: http://localhost:3000 .

```
location / {
  proxy_pass http://localhost:3000;
}
```

Com o Nginx já configurado e rodando, que tal testar essa integração? Reinicie o Nginx através do comando: `nginx -s reload` e também nossa aplicação via comando `npm start`. Se até agora nenhum problema aconteceu, basta acessar sua aplicação através do novo endereço: http://localhost .

Capítulo 10

Continuando os estudos

Finalmente chegamos ao fim deste livro, e como esta tecnologia esta constantemente em evolução, com certeza é sempre bom se manter atualizado com seus novos recursos, ler e acompanhar novas referências sobre esta plataforma. Com isso listarei algumas excelentes referências para você continuar estudando mais e mais Node.js!

Sites, blogs, fóruns e slides

- Site oficial Node.js - http://nodejs.org
- Github do Node.js - http://github.com/joyent/node
- Documentação do Node.js - http://nodejs.org/api
- Blog do Node.js - http://blog.nodejs.org
- NPM Registry - http://npmjs.org
- Blog da comunidade NodeBR - http://nodebr.com

- Fórum do NodeBR - http://groups.google.com/forum/#!forum/nodebr
- Underground WebDev (autor deste livro) - http://udgwebdev.com
- How to Node - http://howtonode.org
- Scoop.it do Node.js - http://scoop.it/t/nodejs-code
- TJ Holowaychuk (autor do Express, EJS e Mocha) - http://tjholowaychuk.com
- NodeCloud - http://nodecloud.org
- DailyJS - http://dailyjs.com
- NetTuts+ - http://net.tutsplus.com
- Blog metaduck - http://metaduck.com
- Blog Waib - http://www.waib.com.br/Blog
- Blog Caelum - http://blog.caelum.com.br
- Blog Cabaré - http://caba.re
- NodeSchool - http://nodeschool.io
- GUJ (discuta sobre Javascript e Node.js) - http://guj.com.br

Eventos, screencasts, podcasts e cursos

- Node Knockout (Hackaton Node.js inspirado em Rails Rumble) - http://nodeknockout.com
- DevCast - Node.js e MongoDB, O casamento perfeito - http://youtube.com/watch?v=-OpUd1Rov2c
- DevCast - Javascript dos novos tempos - http://youtube.com/watch?v=LlUTi5DM4ws
- GrokPodcast Node.js Parte 1 - http://grokpodcast.com/2011/02/17/episodio-19-node-js-parte-1
- GrokPodcast Node.js Parte 2 - http://grokpodcast.com/2011/02/24/episodio-20-node-js-parte-2

- GrokPodcast Node.js Parte 3 - http://grokpodcast.com/2011/03/03/episodio-21-node-js-parte-3
- Nodecasts (Inspirado no Railscast) - http://nodecasts.net
- SailsCasts (Screencasts do Framework Sails.js) - http://irlnathan.github.io/sailscasts/blog/archives
- Nodetuts screencasts - http://nodetuts.com
- Nodeup podcasts - http://nodeup.com
- Codeschool: Real time with Node.js - http://codeschool.com/courses/real-time-web-with-nodejs
- Tuts+ Courses: Building web apps with Node.js and Express - http://tutsplus.com/course/building-web-apps-in-node-and-express
- Peepcode Node.js screencasts - http://peepcode.com/products/nodejs-i
- Alura: Desenvolva aplicações em tempo real com Socket.IO e Node.JS - http://www.alura.com.br/cursos-online-ferramenta/nodejs-socketio

Enfim, espero que seja de grande utilidade essas referências, tenha bons estudos e obrigado por ler este livro.